A Game-theoretic Approach for Synthesizing Fault-Tolerant Embedded Systems

Chih-Hong Cheng*, Harald Ruess†, Alois Knoll* Christian Buckl†
* *Department of Informatics, Technische Universität München, Germany*
†*fortiss GmbH, Germany*
Email:{chengch,knoll}@in.tum.de, {ruess,buckl}@fortiss.org

Abstract

In this paper, we present an approach for fault-tolerant synthesis by combining predefined patterns for fault-tolerance with algorithmic game solving. A non-fault-tolerant system, together with the relevant fault hypothesis and fault-tolerant mechanism templates in a pool are translated into a distributed game, and we perform an incomplete search of strategies to cope with undecidability. The result of the game is translated back to executable code concretizing fault-tolerant mechanisms using constraint solving. The overall approach is implemented to a prototype tool chain and is illustrated using examples.

I. Introduction

In this paper, we investigate methods to perform automatic fault-tolerant (FT for short) synthesis under the context of embedded systems, where our goal is to generate executable code which can be deployed on dedicated hardware platforms.

Creating such a tool supporting the fully-automated process is very challenging as the inherent complexity is high: bringing FT synthesis from theory to practice means solving a problem consisting of (a) interleaving semantics, (b) timing, (c) fault-tolerance, (d) dedicated features of concrete hardware, and optionally, (e) the code generation framework. To generate tamable results, we first constrain our problem space to some simple yet reasonable scenarios (sec. II). Based on these scenarios we can start system modeling (sec. III) taking into account all above mentioned aspects.

To proceed further, we find it important to observe the approach nowadays to understand the need: for engineers working on ensuring fault-tolerance of a system, once the corresponding fault model is decided, a common approach is to select some fault-tolerant patterns [14] (e.g., fragments of executable code) from a pattern pool. Then engineers must fine-tune these mechanisms, or fill in unspecified information in the patterns to make them work as expected. With the above scenario in mind, apart from generating complete FT mechanisms from specification, our synthesis technique emphasizes automatic selection of predefined FT patterns and automatic tuning such that details (e.g., timing) can be filled without human intervention. This also reduces a potential problem where unwanted FT mechanisms are synthesized due to under-specification. Following the statement, we translate the system model, the fault hypothesis, and the set of available FT patterns into a distributed game [18] (sec. V), and a strategy generated by the game solver can be interpreted as a selection of FT patterns together with guidelines of tuning.

For games, it is known that solving distributed games is, in most cases, undecidable [18]. To cope with undecidability, we restrict ourselves to the effort of finding positional strategies (mainly for reachability games). We argue that finding positional strategies is still practical, as the selected FT patterns may introduce additional memory during game creation. Hence, a positional strategy (by pattern selection) combined with selected FT patterns generates mechanisms using memory. By posing this restriction, the problem of finding a strategy of the game (for control) is NP-Complete (sec. VI), and searching techniques (e.g., SAT translation or combining forward search with BDD) are thus applied to assist the finding of solutions.

The final step of the automated process is to translate the result of synthesis back to concrete implementation: the main focus is to ensure that the newly synthesized mechanisms do not change the implementability of the original system (i.e., the new system is schedulable). Based on our modeling framework, this problem can be translated to a linear constraint system, which can be solved efficiently by existing tools.

To evaluate our methods, we have created our prototype software, which utilizes the model-based approach to facilitate the design, synthesis, and code generation for fault-tolerant embedded systems. We demonstrate two small yet representative examples with our tool for a proof-of-concept (sec. VIII); these examples indicate the applicability of the approach. Lastly, we conclude this paper with an overview of related work (sec. IX) and a brief summary including the flow of our approach (sec. X).

II. Motivating Scenario

A. Adding FT Mechanisms to Resist Message Loss

We give a motivating scenario in embedded systems to facilitate our mathematical definitions. The simple system described in Figure 1 contains two *processes* $\mathcal{A}$, $\mathcal{B}$ and one bidirectional *network* $\mathcal{N}$. Processes $\mathcal{A}$ and $\mathcal{B}$ start executing sequential actions together with a looping period of $100ms$. In each period, $\mathcal{A}$ first reads an input using a sensor to

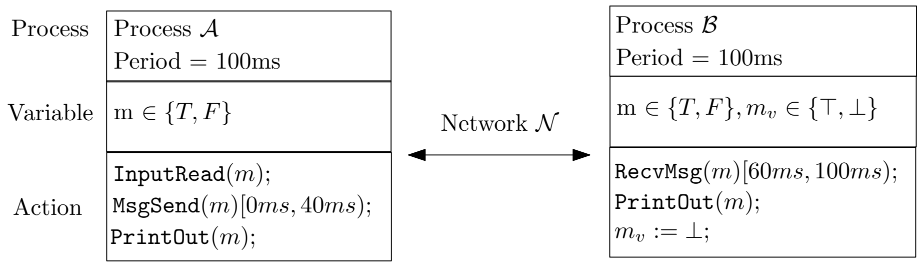


Figure 1. An example for two processes communicating over an unreliable network.

variable m, followed by sending the result to the network $\mathcal{N}$ using the action `MsgSend(m)`, and outputing the value (e.g., to a log).

In process $\mathcal{A}$, for the action `MsgSend(m)`, a message containing value of m is forwarded to $\mathcal{N}$, and $\mathcal{N}$ broadcasts the value to all other processes which contain a variable named m, and set the variable m_v in $\mathcal{B}$ as $\top$ (indicating that the content is valid). However, $\mathcal{A}$ is unaware whether the message has been sent successfully: the network component $\mathcal{N}$ is unreliable, which has a faulty behavior of *message loss*. The fault type and the frequency of the faulty behavior are specified in the *fault model*: in this example for every complete period ($100ms$), at most one message loss can occur.

In $\mathcal{B}$, its first action `RecvMsg(m)` has a property describing an interval $[60, 100)$, which specifies the *release time* and *deadline* of this action to be $60ms$ and $100ms$, respectively. By posing the release time and the deadline, in this example, $\mathcal{B}$ can finalize its decision whether it has received the message m successfully using the equality constraint ($m_v = \bot$), provided that the time interval $[40, 60)$ between (a) deadline of `MsgSend(m)` and (b) release time of `RecvMsg(m)` overestimates the *worst case transmission time* for a message to travel from $\mathcal{A}$ to $\mathcal{B}$. After `RecvMsg(m)`, it outputs the received value (e.g., to an actuator).

Due to the unreliable network, it is easy to observe that two output values may not be the same. Thus the *fault-tolerant synthesis* problem in this example is to perform suitable modification on $\mathcal{A}$ and $\mathcal{B}$, such that two output values from $\mathcal{A}$ and $\mathcal{B}$ are the same at the end of the period, regardless of the disturbance from the network.

B. Solving Fault-Tolerant Synthesis by Instrumenting Primitives

To perform FT synthesis in the example above, our method is to introduce several slots (the size of slots are fixed by the designer) between actions originally specified in the system. For each slot, an atomic operation can be instrumented, and these actions are among the pool of predefined *fault-tolerant primitives*, consisting of message sending, message receiving, local variable modifications, or `null-op`s. Under this setting we have created a game, as the original transitions in the fault-intolerant system combined with all FT primitives available constitute the controller (player-0) moves, and the triggering of faults and the networking can be modeled as environment (player-1) moves.

III. System Modeling

A. Platform Independent System Execution Model

We first define the execution model where timing information is included; it is used for specifying embedded systems and is linked to our code-generation framework. In the definition, for ease of understanding we also give each term intuitive explanations.

Definition 1: Define the syntax of the **Platform-Independent System Execution Model (PISEM)** be $\mathcal{S} = (\mathcal{A}, \mathcal{N}, \mathcal{T})$.

- $\mathcal{T} \in \mathbf{Q}$ is the replication period of the system.
- $\mathcal{A} = \bigcup_{i=1\ldots n_A} \mathcal{A}_i$ is the set of processes, where in $\mathcal{A}_i = (V_i \cup V_{env_i}, \overline{\sigma_i})$,
  - V_i is the set of variables, and V_{env_i} is the set of environment variables. For simplicity assume that V_i and V_{env_i} are of integer domain.
  - $\overline{\sigma_i} := \sigma_1[\alpha_1, \beta_1); \ldots; \sigma_j[\alpha_j, \beta_j); \ldots; \sigma_{k_i}[\alpha_{k_i}, \beta_{k_i})$ is a sequence of actions.
    - $\sigma_j :=$ `send`$(pre, index, n, s, d, v, c) \mid a \leftarrow$ `e` $\mid$ `receive`(pre, c) is an atomic action (action pattern), where
      * $a, c \in V_i$,
      * `e` is function from $V_{env_x} \cup V_i$ to V_i (this includes `null-op`),
      * pre is a conjunction of over equalities/inequalities of variables,
      * $s, d \in \{1, \ldots, n_A\}$ represents the source and destination,
      * $v \in V_d$ is the variable which is expected to be updated in process d,
      * $n \in \{1, \ldots, n_N\}$ is the network used for sending, and
      * $index \in \{1, \ldots, size_n\}$ is the index of the message used in the network.
    - $[\alpha_j, \beta_j)$ is the execution interval, where $\alpha_j \in \mathbf{Q}$ is the release time and $\beta_j \in \mathbf{Q}$ is the deadline.
- $\mathcal{N} = \bigcup_{i=1\ldots n_N} \mathcal{N}_i$, $\mathcal{N}_i = (\mathcal{T}_i, size_i)$ is the set of network.

- $\mathcal{T}_i : \mathbf{N} \rightarrow \mathbf{Q}$ is a function which maps the index (or priority) of a message to the worst case message transmission time (WCMTT).
- $size_i$ is the number of messages used in $\mathcal{N}_i$.

[Example] Based on the above definitions, the system under execution in section II-A can be easily modeled by PISEM: let $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{N}$ in section II-A be renamed in a PISEM as $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{N}_1$. For simplicity, we use $\mathcal{A}.j$ to represent the variable j in process $\mathcal{A}$, assume that the network transmission time is 0, and let v_{env} contain only one variable v in $\mathcal{A}_1$. Then in the modeled PISEM, we have $\mathcal{N}_1 = (f : \mathbf{N} \rightarrow 0, 1)$, $\mathcal{T} = 100$, and the action sequence of process $\mathcal{A}_1$ is

$$m \leftarrow \texttt{InputRead}(v)[0, 40); \texttt{send}(true, 1, 1, 1, 2, m, \mathcal{A}_1.m)[0, 40); v \leftarrow \texttt{PrintOut}(m)[40, 100);$$

For convenience, we use $|\overline{\sigma_i}|$ to represent the length of the action sequence $\overline{\sigma_i}$, $\sigma_j.deadline$ to represent the deadline of σ_j, and $iSet(\overline{\sigma_i})$ to represent a set containing (a) the set of subscript numbers in $\overline{\sigma_i}$ and (b) $|\overline{\sigma_i}|+1$, i.e., $\{1, \ldots, k_i, k_i+1\}$.

Definition 2: The configuration of $\mathcal{S}$ is $(\bigwedge_{i=1\ldots n_A}(v_i, v_{env_i}, \Delta_{next_i}), \bigwedge_{j=1\ldots n_N}(occu_j, s_j, d_j, var_j, c_j, t_j, ind_j), t)$, where

- v_i is the set of the current values for the variable set V_i,
- v_{env_i} is the set of the current values for the variable set V_{env_i},
- $\Delta_{next_i} \in [1, |\overline{\sigma_i}| + 1]$ is the next atomic action index taken in $\overline{\sigma_i}$[1],
- $occu_j \in \{\texttt{false}, \texttt{true}\}$ is for indicating whether the network is busy,
- $s_j, d_j \in \{1, \ldots, n_A\}$,
- $var_j \in \bigcup_{i=1,\ldots,n_A}(V_i \cup V_{env_i})$,
- $c_j \in \mathbf{Z}$ is the content of the message,
- $ind_j \in \{1, \ldots, size_j\}$ is the index of the message occupied in the network,
- t_j is the reading of the clock used to estimate the time required for transmission,
- t is the current reading of the global clock.

The change of configuration is caused by the following operations.

1) (*Execute local action*) For machine i, let s and j be the current configuration for var and Δ_{next_i}, and v_i, v_{env_i} are current values of V_i and V_{env_i}. If $j = |\overline{\sigma_i}| + 1$ then do nothing (all actions in $\overline{\sigma_i}$ have been executed in this cycle); else the action $\sigma_j := var \leftarrow \mathtt{e}[\alpha_j, \beta_j)$ updates var from s to $\mathtt{e}(v_i, v_{env_i})$, and changes Δ_{next_i} to $min\{x | x \in iSet(\overline{\sigma_i}), x > j\}$. This action should be executed between the time interval $t \in [\alpha_j, \beta_j)$.
2) (*Send to network*) For machine i, let s and j be the current configuration for var and Δ_{next_i}. If $j = |\overline{\sigma_i}| + 1$ then do nothing; else the action $\sigma_j := \texttt{send}(pre, index, n, s, d, v, c)[\alpha_j, \beta_j)$ should be processed between the time interval $t \in [\alpha_j, \beta_j)$, and changes Δ_{next_i} to $min\{x | x \in iSet(\overline{\sigma_i}), x > j\}$.
   - When pre is evaluated to true (it can be viewed as an `if` statement), it then checks the condition $occu_n = false$: if the condition holds, it updates network n with value $(occu_n, s_n, d_n, var_n, c_n, t_n, ind_n) := (true, i, d, v, c, 0, index)$. Otherwise it blocks until the condition holds.
   - When pre is evaluated to false, it skips the sending.
3) (*Process message*) For network j, for configuration $(occu_j, s_j, d_j, var, c_j, t_j, ind_j)$ if $occu_j = true$, then during $t_j < \mathcal{T}_j(ind_j)$, a transmission occurs, which updates $occu_j$ to `false`, $A_{d_j}.var$ to c_j, and $A_{d_j}.var_v$ to `true`.
4) (*Receive*) For machine i, let s and j be the current configuration for c and Δ_{next_i}. If $j = |\overline{\sigma_i}| + 1$ then do nothing; else for $\texttt{receive}(pre, c)[\alpha_j, \beta_j)$ in machine i, it is processed between the time interval $t \in [\alpha_j, \beta_j)$ and changes Δ_{next_i} to $min\{x | x \in iSet(\overline{\sigma_i}), x > j\}$[2].
5) (*Repeat Cycle*) When $t = \mathcal{T}$, t is reset to 0, and for all $x \in \{1, \ldots, n_A\}$, Δ_{next_x} are reset to 1.

Notice that by using this model to represent the embedded system under analysis, we make the following assumptions:

- *All processes and networks in $\mathcal{S}$ share a globally synchronized clock.* Note that this assumption can be fulfilled in many hardware platforms, e.g., components implementing the IEEE 1588 [11] protocol.
- For all actions σ, $\sigma.deadline < \mathcal{T}$; for all send actions $\sigma := \texttt{send}(pre, index, n, s, d, v, c)$, $\sigma.deadline + \mathcal{T}_n(index) < \mathcal{T}$, i.e., all processes and networks should finish its work within one complete cycle.

B. Interleaving Model (IM)

Next, we establish the idea of interleaving model (IM) which is used to offer an intermediate representation to bridge PISEM and game solving, such that (a) it captures the execution semantics of PISEM without explicit statements of timing, and (b) by using this model it is easier to connect to the standard representation of games.

Definition 3: Define the syntax of the **Interleaving Model (IM)** be $S_{IM} = (A, N)$.

- $A = \bigcup_{i=1\ldots n_A} A_i$ is the set of processes, where in $A_i = (V_i \cup V_{env_i}, \overline{\sigma_i})$,
  - V_i is the set of variables, and V_{env_i} is the set of environment variables.

[1]Here an interval $[1, |\overline{\sigma_i}| + 1]$ is used for the introduction of FT mechanisms described later.

[2]In our formulation, the $\texttt{receive}(pre, c)$ action can be viewed as a syntactic sugar of `null-op`; its purpose is to facilitate the matching of send-receive pair with variable c.

- $\overline{\sigma_i} := \sigma_1[\wedge_{m=1\ldots n_A}[pc_{1,m_{low}}, pc_{1,m_{up}})]; \ldots; \sigma_j[\wedge_{m=1\ldots n_A}[pc_{j,m_{low}}, pc_{j,m_{up}})]; \ldots; \sigma_{k_i}[\wedge_{m=1\ldots n_A}[pc_{k_i,m_{low}}, pc_{k_i,m_{up}})]$ is a fixed sequence of actions.
  - $\sigma_j := \texttt{send}(pre, index, n, s, d, v, c) \mid \texttt{receive}(pre, c) \mid a \leftarrow \texttt{e}$ is an atomic action, where a, c, e, pre, v, n, s, d are defined similarly as in PISEM.
  - For σ_j, $\forall m \in \{1, \ldots, n_A\}$, $pc_{j,m_{low}}, pc_{j,m_{up}} \in \{1, \ldots, |\overline{\sigma_m}| + 2\}$ is the lower and the upper bound (PC-precondition interval) concerning
    1) precondition of program counter in machine k, when $m \neq i$.
    2) precondition of program counter for itself, when $m = i$.
- $N = \bigcup_{i=1\ldots n_N} N_i$, $N_i = (T_i, size_i)$ is the set of network.
  - $T_i : \mathbf{N} \rightarrow \bigwedge_{m=1\ldots n_A}(\{1, \ldots, |\overline{\sigma_m}| + 2\}, \{1, \ldots, |\overline{\sigma_m}| + 2\})$ is a function which maps the index (or priority) of a message to the PC-precondition interval of other processes.
  - $size_i$ is the number of messages used in $\mathcal{N}_i$.

Definition 4: The configuration of S_{IM} is $(\bigwedge_i(v_i, v_{env_i}, \Delta_{next_i}), \bigwedge_j(occu_j, s_j, d_j, c_j))$, where $v_i, v_{env_i}, \Delta_{next_i}, occu_j, s_j, d_j, c_j$ are defined similarly as in PISEM.

The change of configurations in IM can be interpreted analogously to PISEM; we omit details here but mention three differences:

1) For an action σ_j having the precondition $[\wedge_{m=1\ldots n_A}[pc_{j,m_{low}}, pc_{j,m_{up}})]$, it should be executed between $pc_{j,m_{low}} \leq \Delta_{next_m} < pc_{j,m_{up}}$, for all m.
2) For processing a message, constraints concerning the timing of transmission in PISEM are replaced by referencing the PC-precondition interval of other processes in IM, similar to 1.
3) The system repeats the cycle when $\forall x \in \{1, \ldots, n_A\}$, $\Delta_{next_x} = |\overline{\sigma_x}| + 1$ and $\forall x \in \{1, \ldots, n_N\}$, $occu_x = \texttt{false}$.

IV. GAMES

For the proof of complexity results, we use similar notations in [18] to define a distributed game. Intuitively, distributed games are games formulating multiple processes with no interactions among themselves but only with the environment.

(Local) Games

A *game graph* or *arena* is a directed graph $G = (V_0 \uplus V_1, E)$ whose nodes are partitioned into two classes V_0 and V_1. We only consider the case of two players in the following and call them player 0 and player 1 for simplicity. A *play* starting from node v_0 is simply a maximal path $\pi = v_0 v_1 \ldots$ in G where we assume that player i determines the *move* $(v_k, v_{k+1}) \in E$ if $v_k \in V_i$ ($i \in \{0, 1\}$). With $\text{Occ}(\pi)$ we denote the set of nodes visited by a play π. A *winning condition* defines when a given play π is *won* by player 0; if π is not won by player 0, it is won by player 1. A node v is won by player i if player i can always choose his moves in such a way that he wins any resulting play starting from v.

Distributed Games

We use notations by Mohalik and Walukiewicz [18] to define a distributed game. From now on we call the a game graph defined in sec. IV a *local game graph*.

Definition 5: For all $i \in \{1, \ldots, n\}$, let $G_i = (V_{0_i} \uplus V_{1_i}, E_i)$ be a local game graph with the restriction that it is bipartite. Define a *distributed game* to be $\mathcal{G} = (\mathcal{V}_0 \uplus \mathcal{V}_1, \mathcal{E}, Acc \subseteq (\mathcal{V}_0 \uplus \mathcal{V}_1)^\omega)$:

- $\mathcal{V}_1 = V_{1_1} \times \ldots \times V_{1_n}$ is the set of player 1 (environment) vertices.
- $\mathcal{V}_0 = (V_{0_1} \uplus V_{1_1}) \times \ldots \times (V_{0_n} \uplus V_{1_n}) \setminus \mathcal{V}_1$ is the set of player 0 (control) vertices.
  - For a vertex $x = (x_1, \ldots, x_n)$, we use the function $proj(x, i)$ to retrieve the i-th component x_i, and use $proj(X, i)$ to retrieve the i-th component for a set of vertices X.
- Let $(x_1, \ldots, x_n), (x'_1, \ldots, x'_n) \in \mathcal{V}_0 \uplus \mathcal{V}_1$, then define $\mathcal{E}$ as follows:
  - If $(x_1, \ldots, x_n) \in \mathcal{V}_0$, $((x_1, \ldots, x_n), (x'_1, \ldots, x'_n)) \in \mathcal{E}$ if and only if $\forall i.(x_i \in V_{0_i} \rightarrow (x_i, x'_i) \in E_i) \wedge \forall j.\ (x_j \in V_{1_j} \rightarrow x_j = x'_j)$.
  - For $(x_1, \ldots, x_n) \in \mathcal{V}_1$, if $((x_1, \ldots, x_n), (x'_1, \ldots, x'_n)) \in \mathcal{E}$, then for every x_i, either $x_i = x'_i$ or $x'_i \in V_{0_i}$, and moreover $(x_1, \ldots, x_n) \neq (x'_1, \ldots, x'_n)$.
- Acc is the acceptance condition.

In a distributed game $\mathcal{G} = (\mathcal{V}_0 \uplus \mathcal{V}_1, \mathcal{E}, Acc)$, a play is defined analogously as defined in local games: a *play* starting from node v_0 is a maximal path $\pi = v_0 v_1 \ldots$ in $\mathcal{G}$ where player i determines the *move* $(v_k, v_{k+1}) \in \mathcal{E}$ if $v_k \in \mathcal{V}_i$ ($i \in \{0, 1\}$).

A *distributed strategy* of a distributed game for player 0 is a tuple of functions $\xi = \langle f_1, \ldots, f_n \rangle$, where each function $f_i : (V_{0_i} \uplus V_{1_i})^* \times V_{0_i} \rightarrow (V_{0_i} \uplus V_{1_i})$ is a local strategy which decides the updated location of the local game i based on (a) its observable history of local game i and (b) current position of local game i. Lastly, we call a distributed strategy

Algorithm 1: GeneratePreconditionPC

Data: PISEM model $\mathcal{S} = (\mathcal{A}, \mathcal{N}, \mathcal{T})$
Result: Two maps $mapLB$, $mapUB$ which map from an action σ (or a msg processing by network) to two integer arrays $lower[1 \ldots n_A]$, $upper[1 \ldots n_A]$
begin
/* Initial the map for recording the lower and upper bound for action */
for *action* σ_k *in* $\mathcal{A}_i$ *of* $\mathcal{A}$ **do**
$mapLB.put(\sigma_k$, **new int**$[1 \ldots n_A](1))$ /* Initialize to 1 */
$mapUB.put(\sigma_k$, **new int**$[1 \ldots n_A])$
for $\mathcal{A}_j \in \mathcal{A}$ **do** $mapUB.get(\sigma_k)$[j] := $|\overline{\sigma_j}| + 2$ /* Initialize to upperbound */
$mapLB.get(\sigma_k)$[i] = k; $mapUB.get(\sigma)$[i] = k+1; /* self PC */
for *action* σ_m *in* $\mathcal{A}_i$ *of* $\mathcal{A}$, $m = 1, \ldots, |\overline{\sigma_i}|$ **do**
for *action* σ_n *in* $\mathcal{A}_j$ *of* $\mathcal{A}$, $n = 1, \ldots, |\overline{\sigma_j}|$, $j \neq i$ **do**
1 **if** $\sigma_m.releaseTime > \sigma_n.deadline$ **then**
$mapLB.get(\sigma_m)$[j] := **max**$\{mapLB.get(\sigma_m)$[j], $n+1\}$
2 **if** $\sigma_m.deadline < \sigma_n.releaseTime$ **then**
$mapUB.get(\sigma_m)$[j] := **min**$\{mapUB.get(\sigma_m)$[j], $n+1\}$;
/* Initialize the map for recording the lower and upper bound for msg transmission */
for *action* $\sigma_k = send(pre, ind, n, s, d, v, c)$ *in* $\mathcal{A}_i$ *of* $\mathcal{A}$ **do**
$mapLB.put(n.ind$, **new int**$[1 \ldots n_A](1))$ /* Initialize to 1 */
$mapLB.get(n.ind)$[i] := k+1 /* Strictly later than executing `send()` */
$mapUB.put(n.ind$, **new int**$[1 \ldots n_A])$
for $\mathcal{A}_j \in \mathcal{A}$ **do** $mapUB.get(n.ind)$[j] := $|\overline{\sigma_j}| + 2$ /* Initialize to upperbound */
for *action* $\sigma_k = send(pre, ind, n, s, d, v, c)$ *in* $\mathcal{A}_i$ *of* $\mathcal{A}$ **do**
for *action* σ_m *in* $\mathcal{A}_j$ *of* $\mathcal{A}$, $n = 1, \ldots, |\overline{\sigma_j}|$ **do**
3 **if** $\sigma_k.releaseTime + 0 > \sigma_m.deadline$ **then**
$mapLB.get(n.ind)$[j] := **max**$\{mapLB.get(n.ind)$[j], $m+1\}$
4 **if** $\sigma_k.deadline + \mathcal{T}_n(ind) < \sigma_m.releaseTime$ **then**
$mapUB.get(n.ind)$[j] := **min**$\{mapUB.get(n.ind)$[j], $m+1\}$;

positional, if f_i is a function mapping from V_{0_i} to $V_{0_i} \uplus V_{1_i}$, i.e., the update of location depends only on the current position of local game.

Definition 6: A distributed game $\mathcal{G} = (\mathcal{V}_0 \uplus \mathcal{V}_1, \mathcal{E}, Acc)$ is reachability-winning by a distributed strategy $\xi = \langle f_1, \ldots, f_n \rangle$ over initial states $V_{ini} \in \mathcal{V}_0 \uplus \mathcal{V}_1$ and target states $V_{goal} \in \mathcal{V}_0 \uplus \mathcal{V}_1$, when the following conditions hold:

- $Acc = \{v_0 v_1 \ldots \in (\mathcal{V}_0 \uplus \mathcal{V}_1)^\omega \mid \mathrm{Occ}(v_0 v_1 \ldots) \cap V_{goal} \neq \emptyset\}$.
- For every play $\pi = v_0 v_1 v_2, \ldots$ where $v_0 \in V_{ini}$, player 0 wins π when the following constraints hold:
  - $\pi \in Acc$.
  - $\forall i \in \mathbf{N}_0.\ (v_i \in \mathcal{V}_0 \rightarrow (\forall j \in \{1, \ldots, n\}.\ (proj(v_i, j) \in V_{0_j} \rightarrow proj(v_{i+1}, j) = proj(f_j(v_i), j))))$.

V. Step A: Front-end Translation from Models to Games

A. Step A.1: From PISEM to IM

To translate from PISEM to IM, the key is to generate abstractions from the release time and the deadline information specified in PISEM. As in our formulation, the system is equipped with a globally synchronized clock, the execution of actions respecting the release time and the deadline can be translated into a partial order. Algorithm 1 concretizes this idea by generating PC-intervals in all machines as

- temporal preconditions for an action to execute, or
- temporal preconditions for a network to finish its message processing, i.e., to update a variable in the destination process with the value in the message[3].

[3]Here we assume that in each period, for all $\mathcal{N}_j$, each message of type $ind \in \{1, \ldots, size_j\}$ is sent at most once. In this way, the algorithm can assign an unique PC-precondition interval for every message type.

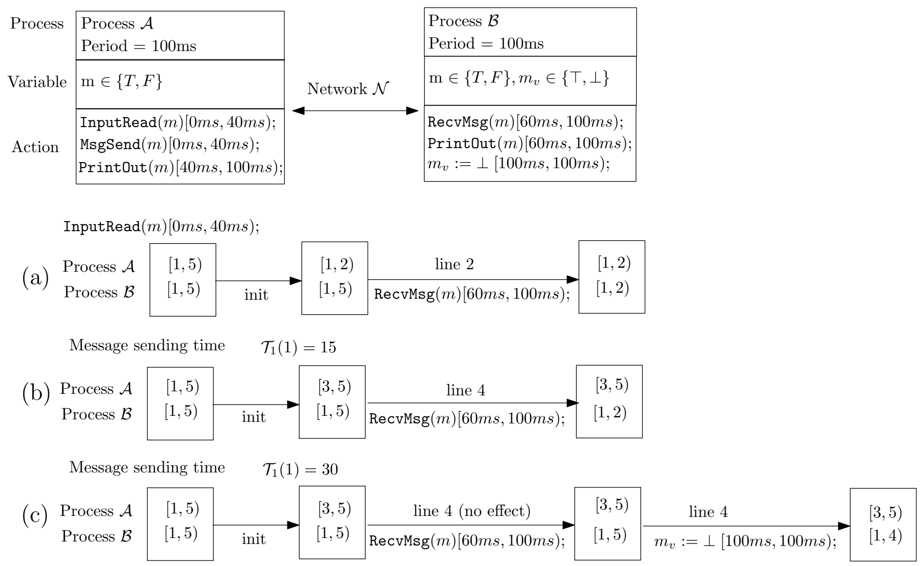


Figure 2. An illustration for Algorithm 1.

Starting from the initialization where no PC is constrained, the algorithm performs a restriction process using four if-statements $\{$**(1)**, **(2)**, **(3)**, **(4)**$\}$ listed.

- In (1), if $\sigma_m.releaseTime > \sigma_n.deadline$, then before σ_m is executed, σ_n should have been executed.
- In (2), if $\sigma_m.deadline < \sigma_n.releaseTime$, then σ_n should not be executed before executing σ_m.
- Similar analysis is done with (3) and (4). However, we need to consider the combined effect together with the network transmission time: we use 0 to represent the best case, and $\mathcal{T}_n(ind)$ for the worst case.

[Example] For the example in sec. II, consider the action $\sigma_1 := m \leftarrow \texttt{InputRead}(v)[0, 40)$ in $\mathcal{A}_1$ of a PISEM. Algorithm 1 returns $mapLB(\sigma)$ and $mapUB(\sigma)$ with two arrays $[1, 1]$ and $[2, 2]$, indicated in Figure 2a. Based on the definition of IM, σ_1 should be executed with the temporal precondition that no action in $\mathcal{A}_2$ is executed, satisfying the semantics originally specified in PISEM. For the analysis of message sending time, two cases are listed in Figure 2b and Figure 2c, where the WCMTT is estimated as 15ms and 30ms, respectively.

B. Step A.2: From IM to Distributed Game

Here we give main concepts how a game is created after step A.1 is executed. To create a distributed game from a given interleaving model $S_{IM} = (A, N)$, we need to proceed with the following three steps:

1) Step A.2.1: Creating non-deterministic timing choices for existing actions: During the translation from a PISEM $\mathcal{S} = (\mathcal{A}, \mathcal{N}, \mathcal{T})$ to its corresponding IM $S_{IM} = (A, N)$, for all process $\mathcal{A}_i$ in $\mathcal{A}$, for every action $\sigma[\alpha, \beta)$ where $\sigma[\alpha, \beta) \in \overline{\sigma_i}$, algorithm 1 creates the PC-precondition interval $[\wedge_{m=1\ldots n_A}[pc_{m_{low}}, pc_{m_{up}})]$ of other processes. Thus in the corresponding game, for $\sigma[\wedge_{m=1\ldots n_A}[pc_{m_{low}}, pc_{m_{up}})]$, each element $\sigma[\wedge_{m=1\ldots n_A}(pc_m)]$, where $pc_{m_{low}} \leq pc_m < pc_{m_{up}}$, is a nondeterministic transition choice which can be selected separately by the game engine.

2) Step A.2.2: Introducing fault-tolerant choices as $\sigma_{\frac{a}{b}}$: In our framework, fault-tolerant mechanisms are similar to actions, which consist of two parts: *action pattern* σ and *timing precondition* $[\wedge_{m=1\ldots n_A}[pc_{m_{low}}, pc_{m_{up}})]$. Compared to existing actions where nondeterminism comes from timing choices, for fault-tolerance transition choices include all combinations from (1) timing precondition and (2) action patterns available from a predefined pool.

We use the notation $\sigma_{\frac{a}{b}}$, where $\frac{a}{b} \in \mathbf{Q} \backslash \mathbf{N}$, to represent an inserted action pattern between $\sigma_{\lfloor \frac{a}{b} \rfloor}$ and $\sigma_{\lceil \frac{a}{b} \rceil}$. With this formulation, multiple FT mechanisms can be inserted within two consecutive actions σ_i, σ_{i+1} originally in the system, and the execution semantic follows what has been defined previously: as executing an action updates Δ_{next_i} to $min\{x | x \in iSet(\overline{\sigma_i}), x > j\}$, updating to a rational value is possible. Note that as $\sigma_{\frac{a}{b}}$ is only a fragment without temporal preconditions, we use algorithm 2 to generate all possible temporal preconditions satisfying the semantics of the original interleaving model: after the synthesis only temporal conditions satisfying the acceptance condition will be chosen.

We conclude this step with two remarks:

- For all existing actions, the non-deterministic choice generation in step A.2.1 must be modified to contain these rational points introduced by FT mechanisms.
- A problem induced by FT synthesis is whether the system behavior changes due to the introduction of FT mechanisms. We answer the problem by splitting into two subproblems:

Algorithm 2: DecideInsertedFTTemplateTiming

Data: $\sigma_c[\wedge_{m=1\dots n_A}[pc_{c,m_{low}}, pc_{c,m_{up}})]$, $\sigma_d[\wedge_{m=1\dots n_A}[pc_{d,m_{low}}, pc_{d,m_{up}})]$, which are consecutive actions in $\overline{\sigma_i}$ of A_i of $S_{IM} = (A, N)$, and one newly added action pattern $\sigma_{\frac{a}{b}}$ to be inserted between

Result: Temporal preconditions for action pattern $\sigma_{\frac{a}{b}}$: $[\wedge_{m=1\dots n_A}[pc_{\frac{a}{b},m_{low}}, pc_{\frac{a}{b},m_{up}})]$

begin

  for $m = 1, \dots, n_A$ **do**

    if $m \neq i$ **then**

      $pc_{\frac{a}{b},m_{low}} := pc_{c,m_{low}}$ /* Use the lower bound of c for its lower bound */

      $pc_{\frac{a}{b},m_{up}} := pc_{d,m_{up}}$ /* Use the upper bound of d for its upper bound */

    else

      $pc_{\frac{a}{b},m_{low}} := \frac{a}{b}$; $pc_{\frac{a}{b},m_{up}} := d$

	DG	SDG
State space	product of all vertices in local games	product of all variables (including variables used in local games)
Vertex partition (V_0 and V_1)	explicit partition	use $pred$ to perform partition
Player-0 transitions	defined in local games	defined in $\overline{\sigma_i}$ of A_i, for all $i \in \{1, \dots, n_A\}$
Player-1 transitions	explicitly specified in the global game	defined in N and σ_f

Figure 3. Comparison between DG and SDG

- **[Problem 1]** Whether the system is still schedulable due to the introduction of FT actions, as these FT actions also consume time. This can only be answered when the result of synthesis is generated, and we leave this to section VII.
- **[Problem 2]** Whether the networking behavior remains the same. This problem *must* be handled before game creation, as introducing a FT message may significantly influence the worst case message transmission time (WCMTT) of all existing messages, leading a completely different networking behavior. The answer of this problem depends on many factors, including the hardware in use, the configuration setting, and the analysis technique used for the estimation of WCMTT. In Appendix A we give a simple analysis for ideal CAN buses [7], which are used most extensively in industrial and automotive embedded systems: in the analysis, we propose conditions where newly added messages do ***not*** change the existing networking behavior. Similar analysis can be done with other timing-predictable networks, e.g., FlexRay [20].

3) Step A.2.3: Game Creation by Introducing Faults: In our implementation, we do not generate the primitive form of distributed games (DG), as the definition of DG is too primitive to manipulate. Instead, algorithms in our implementations are based on our created variant called ***symbolic distributed games (SDG)***:

Definition 7: Define a symbolic distributed game $\mathcal{G}_{ABS} = (V_f \uplus V_{CTR} \uplus V_{ENV}, A, N, \sigma_f, pred)$.

- V_f, V_{CTR}, V_{ENV} are disjoint sets of (fault, control, environment) variables.
- $pred : V_f \times V_{CTR} \times V_{ENV} \rightarrow \{\texttt{true}, \texttt{false}\}$ is the partition condition.
- $A = \bigcup_{i=1\dots n_A} A_i$ is the set of **symbolic local games (processes)** , where in $A_i = (V_i \cup V_{env_i}, \overline{\sigma_i})$,
  - V_i is the set of variables, and $V_{env_i} \subseteq V_{ENV}$.
  - $\overline{\sigma_i} := \bigcup \sigma_{i_1}\langle \wedge_{m=1,\dots,n_A} pc_{i_{1_m}}\rangle; \dots; \bigcup \sigma_{i_k}\langle \wedge_{m=1,\dots,n_A} pc_{i_{k_m}}\rangle$ is a sequence, where $\forall j = 1, \dots, k$, $\bigcup \sigma_{i_j}\langle \wedge_{m=1,\dots,n_A} pc_{i_{j_m}}\rangle$ is a set of choice actions for player-0 in A_i.
    - σ_{i_j} is defined similarly as in IM.
    - $\forall m = \{1, \dots, n_A\}$, $pc_{i_{j_m}} \in [pc_{i_j,m_{low}}, pc_{i_j,m_{up}})$, $pc_{i_j,m_{low}}, pc_{i_j,m_{up}} \in iSet(\overline{\sigma_m})$.
  - $V_{CTR} = \bigcup_{i=1\dots n_A} V_i$.
- $N = \bigcup_{i=1\dots n_N} N_i$, $N_i = (T_i, size_i, tran_i)$ is the set of network processes.
  - T_i and $size_i$ are defined similarly as in IM.
  - $tran_i : V_f \times (\{\texttt{true}, \texttt{false}\} \times \{1, \dots, n_A\}^2 \times \bigcup_{i=1,\dots,n_A}(V_i \cup V_{env_i}) \times \mathbf{Z} \times \{1, \dots, size_i\}) \rightarrow V_f \times (\{\texttt{true}, \texttt{false}\} \times \{1, \dots, n_A\}^2 \times \bigcup_{i=1,\dots,n_A}(V_i \cup V_{env_i}) \times \mathbf{Z} \times \{1, \dots, size_i\})$ is the network transition relation for processing messages (see sec. III-A for meaning), but can be influenced by additional variables in V_f.
- $\sigma_f : V_f \times V_{CTR} \times V_{ENV} \times \bigwedge_{i=1\dots n_A} iSet(\overline{\sigma_i}) \rightarrow V_{ENV} \times V_f \times \bigwedge_{i=1\dots n_A} iSet(\overline{\sigma_i})$ is the environment update relation.

We establish an analogy between SDG and DG using Figure 3.

1) The configuration v of a SDG is defined as the product of all variables used.
2) A play for a SDG starting from state v_0 is a maximal path $\pi = v_0 v_1 \dots$, where

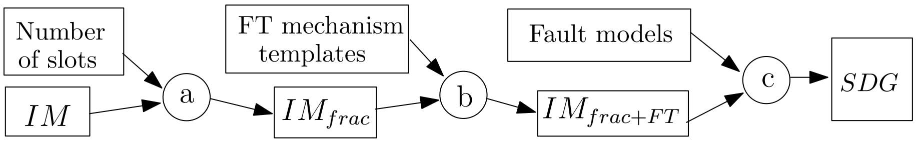


Figure 4. Creating the SDG from IM, FT mechanisms, and faults.

- In v_k, player-1 determines the move $(v_k, v_{k+1}) \in E$ when $pred(v_k)$ is evaluated to `true` (`false` for player-0); the partition of vertices V_0 and V_1 in a SDG is implicitly defined based on this, rather than specified explicitly as in a distributed game.
- A move (v_k, v_{k+1}) is a selection of executable transitions defined in N, σ_f, or A; in our formulation, transitions in N and σ_f are all environment moves[4], while transitions in A are control moves[5].

3) Lastly, a distributed positional strategy for player-0 in a SDG can be defined analogously as to uniquely select an action from the set $\bigcup \sigma_{\alpha_j} \langle \wedge_{m=1,\dots,n_A}, pc_{\alpha_{j_m}} \rangle$, for all A_i and for all program counter j defined in $\overline{\sigma_i}$. Each strategy should be insensitive of contents in other symbolic local games.

We now summarize the logical flow of game creation using Figure 4.

- (a) Based on the fixed number of slots (for FT mechanisms) specified by the user, extend IM to IM_{frac} to contain fractional PC-values induced by the slot.
- (b) Create $IM_{frac+FT}$, including the sequence of choice actions (as specified in the SDG) by
  - Extracting action sequences defined in IM_{frac} to choices (step A.2.1).
  - Inserting FT choices (step A.2.2).
- (c) Introduce faults and partition player-0 and player-1 vertices: In engineering, a *fault model* specifies potential undesired behavior of a piece of equipment, such that engineers can predict the consequences of system behavior. Thus, a *fault* can be formulated with three tuples[6]:
  1) The fault type (an unique identifier, e.g., `MsgLoss`, `SensorError`).
  2) The maximum number of occurrences in each period.
  3) Additional transitions not included in the original specification of the system (*fault effects*).

  We perform the translation into a game using the following steps.
  - For (1), introduce variables to control the triggering of faults.
  - For (2), introduce counters to constrain the maximum number of fault occurrences in each period.
  - For (3), for each transition used in the component influenced by the fault, create a corresponding fault transition which is triggered by the variable and the counter; similarly create a transition with normal behavior (also triggered by the variable and the counter). Notice that our framework is able to model faults actuating on the FT mechanisms, for instance, the behavior of network loss on the newly introduced FT messages.

[Example] We outline how a game (focusing on fault modeling) is created with the example in sec. II; similar approaches can be applied for input errors or message corruption; here the modeling of input (for `InputRead(m)`) is skipped.

- Create the predicate $pred$: $pred$ is evaluated to `false` in all cases except (a) when the boolean variable *occu* (representing the network occupance) is evaluated to `true` and (b) when for all $i \in \{1, \dots, n_A\}$, $\Delta_{next_i} = |\overline{\sigma_i}| + 1$ (end of period); the predicate partitions player-0 and player-1 vertices.
- For all process i and program counter j, the set of choice actions $\bigcup \sigma_{\alpha_j} \langle \wedge_{m=1,\dots,n_A}, pc_{\alpha_{j_m}} \rangle$ are generated based on the approach described previously.
- Create variable $v_f \in V_f$, which is used to indicate whether the fault (`MsgLoss`) has been activated in this period.
- In this example, as the maximum number of fault occurrences in each period is 1, we do not need to create additional counters.
- For each message sending transition t in the network, create two normal transitions $(v_f = \texttt{true} \wedge v'_f = \texttt{true}) \wedge t$ and $(v_f = \texttt{false} \wedge v'_f = \texttt{false}) \wedge t$ in the game.
- For each message sending transition t in the network, generate a transition t' where the message is sent, but the value is not updated in the destination. Create a fault transition $(v_f = \texttt{false} \wedge v'_f = \texttt{true}) \wedge t'$ in the game.
- Define σ_f to control v_f: if for all $i \in \{1, \dots, n_A\}$, $\Delta_{next_i} = |\overline{\sigma_i}| + 1$, then update v_f to `false` as Δ_{next_i} updates to 1 (reset the fault counter at the end of the period).

VI. Step B: Solving Distributed Games

We summarize the result from [18] as a general property of distributed games.

[4]As the definition of distributed games features multiple processes having no interactions among themselves but only with the environment, a SDG is also a distributed game. In the following section, our proof of results and algorithms are all based on DG.

[5] This constraint can be released such that transitions in A can either be control (normal) or environment (induced by faults) moves; here we leave the formulation as future work.

[6]For complete formulation of fault models, we refer readers to our earlier work [6].

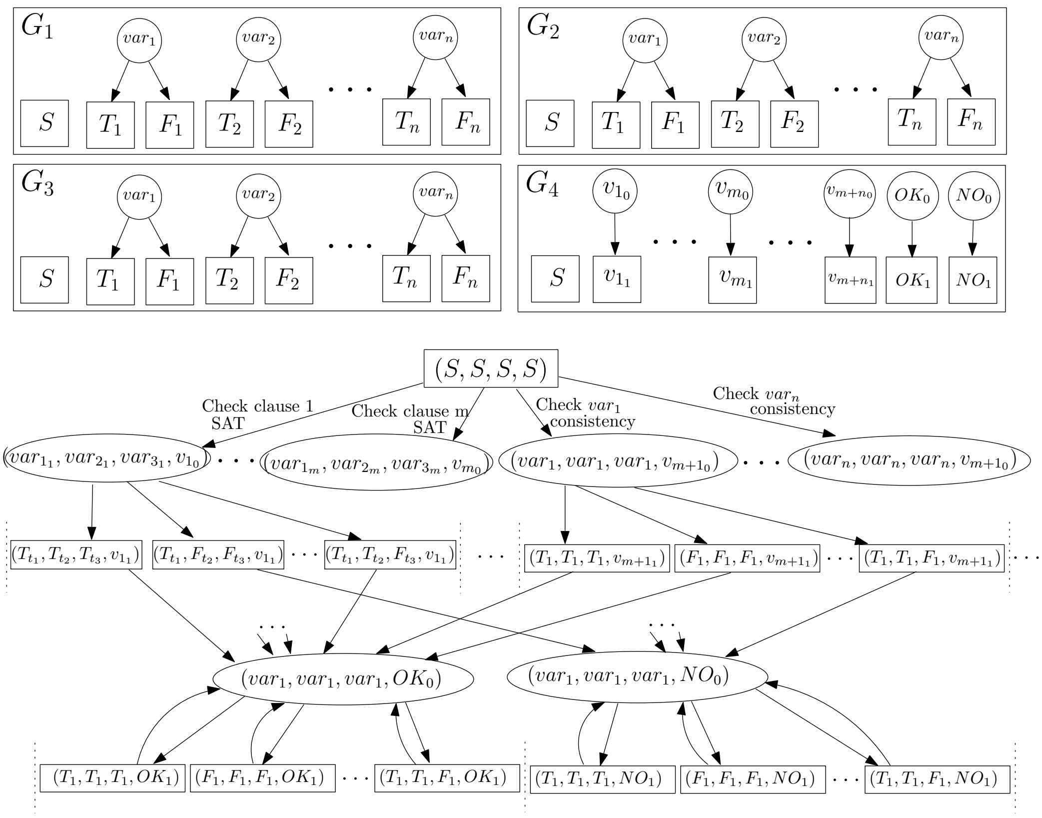


Figure 5. Illustrations for the reduction from 3SAT to $PositionalDG_0$.

Theorem 1: There exists distributed games with global winning strategy but (a) without distributed memoryless strategies, or (b) all distributed strategies require memory. In general, for a finite distributed game, it is undecidable to check whether a distributed strategy exists from a given position [18].

As the problem is undecidable in general, we restrict our interest in finding a distributed positional strategy for player 0, if there exists one. We also focus on games with reachability winning conditions. By posing the restriction, the problem is NP-Complete.

Theorem 2: **[**$PositionalDG_0$**]** Given a distributed game $\mathcal{G} = (\mathcal{V}_0 \uplus \mathcal{V}_1, \mathcal{E})$, an initial state $x = (x_1, \ldots, x_n)$ and a target state $t = (t_1, \ldots, t_n)$, deciding whether there exists a positional (memoryless) distributed strategy for player-0 from x to t is NP-Complete.

Proof:

We first start by recalling the definition of attractor, a term which is commonly used in the game and later applied in the proof. Given a game graph $G = (V_0 \uplus V_1, E)$, for $i \in \{0, 1\}$ and $X \subseteq V$, the map $\text{attr}_i(X)$ is defined by

$$\text{attr}_i(X) := X \cup \{v \in V_i \mid vE \cap X \neq \emptyset\} \cup \{v \in V_{1-i} \mid \emptyset \neq vE \subseteq X\},$$

i.e., $\text{attr}_i(X)$ extends X by all those nodes from which either player i can move to X within one step or player $1-i$ cannot prevent to move within the next step. (vE denotes the set of successors of v.) Then $\text{Attr}_i(X) := \bigcup_{k \in \mathbf{N}} \text{attr}_i^k(X)$ contains all nodes from which player i can force any play to visit the set X.

We continue our argument as follows.

[NP] The reachability problem for a distributed game can be solved in NP: a solution instance $\xi = \langle f_1, \ldots, f_n \rangle$ is a strategy which selects exactly one edge for every control vertex in the local game. As the distributed game graph is known, after the selection we calculate the reachability attractor $\text{Attr}_0(\{t\})$ of the distributed game: during the calculation we overlook transitions which is not selected (in the strategy) in the local game. This means that in the distributed game, to add a control vertex $v \in \mathcal{V}_0$ to the attractor using the edge (v, u), we must ensure that $\forall j \in \{1, \ldots, n\}. \ (proj(v_i, j) \in V_{0_j} \rightarrow proj(u, j) = proj(f_j(v), j))$. Lastly, we check if the initial state is contained; the whole calculation and checking process can be done in deterministic P-time.

[NP-C] For completeness proof, we perform a reduction from 3SAT to the finding of positional strategies in a distributed game. Given a set of 3CNF clauses $\{C_1, \ldots, C_m\}$ under the set of literals $\{var_1, \overline{var_1}, \ldots, var_n, \overline{var_n}\}$ and variables $\{var_1, \ldots, var_n\}$, the distributed game $\mathcal{G}$ is created as follows (see Figure 5 for illustration):

- Create 3 local games G_1, G_2, and G_3, where for $G_i = (V_{0_i} \uplus V_{1_i}, E_i)$:

- $V_{0_i} = \{var_1, \dots, var_n\}$, $V_{1_i} = \{S, T_{var_1}, F_{var_1}, \dots, T_{var_n}, F_{var_n}\}$.
- $E_i = \bigcup_{j=1,\dots,n}\{(var_j, T_{var_j}), (var_j, F_{var_j})\}$.
- Create local game $G_4 = (V_{0_4} \uplus V_{1_4}, E_4)$:
- $V_{0_4} = \{OK_0, NO_0\} \cup \bigcup_{j=1,\dots,m+n}\{v_{j_0}\}$.
- $V_{1_4} = \{S, OK_1, NO_1\} \cup \bigcup_{j=1,\dots,m+n}\{v_{j_1}\}$.
- $E_4 = \bigcup_{j=1,\dots,m+n}\{(v_{j_0}, v_{j_1})\} \cup \{(OK_0, OK_1), (NO_0, NO_1)\}$.
- Second, create the distributed game $\mathcal{G}$ from local games above, and define the set of environment transition to include the following types using the 3SAT problem:
  1) (Intention to check SAT) In the 3SAT problem, for clause $C_i = (l_{1_i} \vee l_{2_i} \vee l_{3_i})$, let the variable for literals $l_{1_i}, l_{2_i}, l_{3_i}$ be $var_{1_i}, var_{2_i}, var_{3_i}$. Create a transition in the distributed game from (S, S, S, S) to $(var_{1_i}, var_{2_i}, var_{3_i}, v_{i_0})$.
  2) (Intention to check consistency) In the 3SAT problem, for variable var_i, Create a transition in the distributed game from (S, S, S, S) to $(var_i, var_i, var_i, v_{m+i_0})$.
  3) (Result of clause) In the 3SAT problem, for clause $C_i = (l_{1_i} \vee l_{2_i} \vee l_{3_i})$, let the variable for the clause be $var_{1_i}, var_{2_i}, var_{3_i}$. We refer the vertex evaluating var_{j_i} as `true` to T_i in the local game G_j; similarly, we use F_i for a variable being evaluated `false`. For each clause C_i, enumerate over 8 cases for the assignments of $var_{1_i}, var_{2_i}, var_{3_i}$ which make C_i true.
     a) For cases which makes the assignment true, create an edge from the assignment to $(var_1, var_1, var_1, OK_0)$; for example, if $var_{1_i} = \texttt{true}, var_{2_i} = \texttt{false}, var_{3_i} = \texttt{true}$ makes a satisfying assignment to C_i, create an edge $((T_{1_i}, F_{2_i}, T_{3_i}, v_{i_1}), (var_1, var_1, var_1, OK_0))$.
     b) For cases which makes the assignment false, create an edge from the assignment to $(var_1, var_1, var_1, NO_0)$.
  4) (Result of variable consistency) For all $i \in \{1, \dots, n\}$:
     a) Create two edges $((T_i, T_i, T_i, v_{m+i_1}), (var_1, var_1, var_1, OK_0))$ and $((F_i, F_i, F_i, v_{m+i_1}), (var_1, var_1, var_1, OK_0))$.
     b) For other 6 combinations $(T_i, F_i, F_i, v_{m+i_1}), (F_i, F_i, T_i, v_{m+i_1})$, $(T_i, F_i, F_i, v_{m+i_1}), (T_i, T_i, F_i, v_{m+i_1}), (F_i, T_i, T_i, v_{m+i_1})$, $(T_i, F_i, T_i, v_{m+i_1})$, create edges to $(var_1, var_1, var_1, NO_0)$.
  5) (Continuous execution) For all $i \in \{1, \dots, n\}$:
     a) For all combinations $(T_i, F_i, F_i, OK_1), (F_i, F_i, T_i, OK_1), (T_i, F_i, F_i, OK_1)$, $(F_i, F_i, T_i, OK_1), (T_i, F_i, F_i, OK_1), (T_i, T_i, F_i, OK_1), (F_i, T_i, T_i, OK_1)$, (T_i, F_i, T_i, OK_1), create edges to $(var_1, var_1, var_1, OK_0)$.
     b) For all combinations $(T_i, F_i, F_i, NO_1), (F_i, F_i, T_i, NO_1), (T_i, F_i, F_i, NO_1)$, $(F_i, F_i, T_i, NO_1), (T_i, F_i, F_i, NO_1), (T_i, T_i, F_i, NO_1), (F_i, T_i, T_i, NO_1)$, (T_i, F_i, T_i, NO_1), create edges to $(var_1, var_1, var_1, NO_0)$.

We claim that $\{C_1, \dots, C_m\}$ is satisfiable iff $\mathcal{G}$ has a positional distributed strategy to reach $(var_1, var_1, var_1, OK_0)$ from (S, S, S, S).

1) If $\{C_1, \dots, C_m\}$ is satisfiable, let the set of satisfying literals be L', and assume that for all literals, in each pair $(var_i, \overline{var_i})$ exactly one of them is in L' (this is always possible). For the distributed game $\mathcal{G}$, in local games G_1, G_2 and G_3, let the positional strategy for control vertex var_i move to T_i if $var_i \in L'$, and move to F_i if $\overline{var_i} \in L'$ (for G_4, simply use the local edge). In a play, as player-1 starts the move, any of his selection leads to a player-0 vertex:
   - If player-1 choose edges of type 1 (intension to check the clause of SAT), for G_1, G_2 and G_3, the vertex uses its positional strategy, which corresponds to the assignment in the clause. The combined move then forces player-1 to choose an edge of type 3(a), leading to the target state.
   - If player-1 choose edges of type 2 (intension to check the consistency), as the positional strategies for G_1, G_2 and G_3 are all derived from the same satisfying instance of the 3SAT problem, for each strategy, it performs the same move from var_i to T_i or to F_i; the combined move of player-0 forces player-1 to choose an edge of type 4(a), leading to the target state.
2) Consider a distributed positional strategy $\langle f_1, f_2, f_3, f_4 \rangle$ which reaches $(var_1, var_1, var_1, OK_0)$ from (S, S, S, S). In G_1, for each control vertex var_i, it points to T_i or F_i. The positional strategy of G_1 generates a satisfying instance of the 3SAT problem:
   - Assign var_i in the 3SAT problem to `true` if the strategy points vertex var_i in G_1 to T_i.
   - Assign var_i in the 3SAT problem to `false` if the strategy points vertex var_i in G_1 to F_i.

We analyze the size of the game and the time required to perform the reduction.

- For $i = 1, 2, 3$, G_i contains $3n + 1$ vertices, and G_4 has $2(m + n + 2) + 1$ vertices. As the total vertices of the distributed game is the product, it is polynomial to the original 3SAT problem instance.
- Consider the time required to perform reduction from 3SAT to $PositionalDG_0$:

- For $i = 1, 2, 3$, G_i, they are constructed in $\mathcal{O}(n)$.
- G_4 is constructed in $\mathcal{O}(m + n)$.
- For the distributed game, vertices are constructed polynomial to m and n, more precisely $\mathcal{O}(n^3(m + n))$.
- For edges in the distributed game, we consider the most complicated case, i.e. creating an edge of type 3. Yet it takes constant time to check and establish the connection, and for each player-1 vertex except (S, S, S, S) which has $m + n$ edges, at most 8 edges are created. Therefore, the total required time for edge construction is also polynomial to m and n.

Therefore, 3SAT $\leq_{poly}$ $PositioalDG_0$, which concludes the proof. ■

With the NP-completeness proof, finding a distributed reachability strategy for distributed games amounts to the process of searching. For example, it is possible to perform a bounded-depth forward search over choices of local transitions: during the search, the selection of edges is constructed as a tree node in the search tree, and the set of reachable vertices (represented as BDD) based on the selection is also stored in the tree node. This method is currently implemented in our framework.

A. Solving Distributed Games using SAT Methods

Apart from the search method above, in this section we give an alternative approach based on a reduction to SAT. Madhusudan, Nam, and Alur [1] designed the *bounded witness algorithm* (based on unrolling) for solving reachability (local) games. Although based on their experiment, the witness algorihm is not as efficient as the BDD based approach in centralized games, we find this concept potentially useful for solving distributed games. For this, we have created a variation (Algorithm 3) for this purpose.

To provide an intuition, first we paraphrase the concept of witness defined in [1], a set of states which witnesses the fact that player 0 wins. In [1], consider the generated SAT problem from a local game $G = (V_0 \uplus V_1, E)$ trying to reach from V_{init} to V_{goal}: for $i = 1, \ldots, d$ and vertex $v \in V_0 \uplus V_1$, variable $\langle v \rangle_i = \texttt{true}$ when one of the following holds:

1) $v \in V_{init}$ and $i = 1$ (if $v \notin V_{init} \wedge i = 1$ then $\langle v \rangle_i = \texttt{false}$).
2) $v \in V_{goal}$ (if $v \notin V_{goal} \wedge i = d$ then $\langle v \rangle_i = \texttt{false}$).
3) $v \in V_0 \setminus V_{goal}$ and $\exists v' \in V_0 \uplus V_1.\ \exists e \in E.\ \exists j > i.\ (e = (v, v') \wedge \langle v' \rangle_j = \texttt{true})$
4) $v \in V_1 \setminus V_{goal}$ and $\forall e = (v, v') \in E.\ \exists j > i.\ \langle v' \rangle_j = \texttt{true}$

This recursive definition implies that if v in V_0 (resp. in V_1) is not the goal but in the witness set, then exists one (resp. for all) successor v' which should either be (i) in a goal state or (ii) also in the witness: note that for (ii), the number of allowable steps to reach the goal is decreased by one. This definition ensures that all plays defined in the witness reaches the goal from the initial state within $d - 1$ steps: If a play (starting from initial state) has proceeded $d - 1$ steps and reached $u \notin V_{goal}$, then based on (2), $\langle u \rangle_d$ should be $\texttt{false}$. However, based on (1), (3), (4) the $\langle u \rangle_d$ should be set to $\texttt{true}$ (reachable from initial states using $d - 1$ steps). Thus the SAT problem should be unsatisfiable.

In general, Algorithm 3 creates constraints based on the above concept, but compared to the bounded local game reachability algorithm in [1], it contains slight modifications:

1) When a variable $\langle v \rangle_i$ is evaluated to $\texttt{true}$, it means that vertex v can reach the target state within $d - i$ steps, which is the same as what is defined in [1]. However, we introduce more variables for edges in local games, which is shown in STEP 1: when a variable $\langle e \rangle$ is evaluated to $\texttt{true}$, the distributed strategy uses the local transition e.
2) To achieve locality, we must include constraints specified in STEP 4: the positional (memoryless) strategy disallows to change the use of local edges from a given vertex.
3) We modify the impact of control edge selection in STEP 6 by adding an additional implication "$\langle e \rangle \Rightarrow$" over the original constraint in the witness algorithm [1]. Here as in Mohalik and Walukiwitz's formulation, all subgames in a control position should proceed a move (the progress of a global move is a combination of local moves), we need to create constraints considering all possible local edge combinations.

In appendix B, we give an alternate algorithm working with different formulation of distributed games where in each control location, only one local game can move: a run of the game may execute multiple local moves until it reaches a state where all local games are in an environment position. We find this alternative formulation closer to the interleaving semantics of distributed systems.

VII. Conversion from Strategies to Concrete Implementations

Once when the distributed game has returned a positive result, and assume that the result is represented as an IM, the remaining problem is to check whether the synthesized result can be translated to PISEM and thus further to concrete implementation. If for each existing action or newly generated FT mechanism, the worst case execution time is known (with available WCET tools, e.g., AbsInt[7]), then we can always answer whether the system is implementable by a full system rescheduling, which can be complicated. Nevertheless, based on our system modeling (assumption with a globally synchronized clock), perform modification on the release time or the deadline on existing actions from the

[7] http://www.absint.com/

Algorithm 3: PositionalDistributedStrategy_BoundedSAT_0

Data: Distributed game graph $\mathcal{G} = (\mathcal{V}_0 \uplus \mathcal{V}_1, \mathcal{E})$, set of initial states V_{init}, set of target states V_{goal}, the unrolling depth d

Result: Output: whether a distributed positional strategy exists to reach V_{goal} from v_{init}

```
begin
    let clauseList := getEmptyList() /* Store all clauses for SAT solvers */
    /* STEP 1: Variable creation */
    for v = (v_1, ..., v_m) ∈ V_0 ⊎ V_1 do
        create d boolean variables ⟨v_1, ..., v_m⟩_1, ..., ⟨v_1, ..., v_m⟩_d;
    for local control transition e = (x_i, x'_i) ∈ E_i, x_i ∈ V_{0_i} do
        create boolean variable ⟨e⟩;
    /* STEP 2: Initial state constraints */
    for v = (v_1, ..., v_m) ∈ V_0 ⊎ V_1 do
        if (v_1, ..., v_m) ∈ V_init then
            clauseList.add([⟨v_1, ..., v_m⟩_1])
        else
            clauseList.add([¬⟨v_1, ..., v_m⟩_1])
    /* STEP 3: Target state constraints */
    for v = (v_1, ..., v_m) ∈ V_0 ⊎ V_1 do
        if (v_1, ..., v_m) ∈ V_goal then
            clauseList.add([⟨v_1, ..., v_m⟩_1 ∧ ... ∧ ⟨v_1, ..., v_m⟩_d])
        else
            clauseList.add([¬⟨v_1, ..., v_m⟩_d])
    /* STEP 4: Unique selection of local transitions (for distributed positional strategy) */
    for local control transition e = (x_i, x'_i) ∈ E_i, x_i ∈ V_{0_i} do
        for local transition e_1 = (x_i, x'_{i_1}), ..., e_k = (x_i, x'_{i_k}) ∈ E_i, e_1 ... e_k ≠ e do
            clauseList.add([⟨e⟩ ⇒ (¬⟨e_1⟩ ∧ ... ∧ ¬⟨e_k⟩)])
    /* STEP 5: If a control vertex is in the attractor (winning region) but not a goal,
    an edge should be selected to reach the goal state */
    for v = (v_1, ..., v_m) ∈ V_0 do
        for v_i, i = 1, ..., m do
            if v_i ∈ V_{0_i} \ V_goal then
                let ⋃_j e_j be the set of local transitions starting from v_i in G_i
                if ⋃_j e_j ≠ φ then
                    clauseList.add([(⋁_{i=1...d} ⟨v⟩_i) ⇒ (⋁_j ⟨e⟩_j)])
    /* STEP 6: Impact of control edge selection (simultaneous progress) */
    for v = (v_1, ..., v_m) ∈ V_0 do
        forall the edge combination (e_1, ..., e_m): e_i = (v_i, v'_i) ∈ E_i when v_i ∈ V_{0_i} or e_i = (v_i, v_i) when x_i ∈ V_{1_i}
        do
            /* e_i = (v_i, v_i) when x_i ∈ V_{1_i} are simply dummy edges for ease of formulation */
            for j = 1, ..., d − 1 do
                clauseList.add([⟨v_1, ..., v_m⟩_j ⇒ ((⋀_{{i|v_i ∈ V_{0_i}}} ⟨e_i⟩) ⇒ (⟨v'_1, ..., v'_m⟩_{j+1}])
    /* STEP 7: Impact of environment vertex */
    for environment vertex v = (v_1, ..., v_m) ∈ V_1 do
        let the set of successors be ⋃_i v_i; for j = 1, ..., d − 1 do
            clauseList.add([⟨v⟩_j ⇒ (⋀_i (⟨v_i⟩_{j+1} ∨ ... ∨, ⟨v_i⟩_d))]);
    /* STEP 8: Invoke the SAT solver: return true when satisfiable */
    return invokeSATsolver(clauseList)
```

Process	Process $\mathcal{A}$ Period = 100ms		Process $\mathcal{B}$ Period = 100ms
Variable	m $\in \{T, F\}$ req, rsp $\in \{T, F\}, req_v, rsp_v \in \{\top, \bot\}$	Network $\mathcal{N}$ $\longleftrightarrow$	$m \in \{T, F\}, m_v \in \{\top, \bot\}$ req, rsp $\in \{T, F\}, req_v, rsp_v \in \{\top, \bot\}$
Action	$\texttt{InputRead}(m)$; $\texttt{MsgSend}(m)[0ms, 40ms)$; $\texttt{RecvMsg}(req)$; $\texttt{if}(req_v \neq \bot)\ rsp := m$; $\texttt{if}(req_v \neq \bot)\ \texttt{MsgSend}(rsp)$; $\texttt{PrintOut}(m); [99ms, 100ms)$ $req_v := \bot; rsp_v := \bot$;		$\texttt{RecvMsg}(m)[60ms, 99ms)$; $\texttt{if}(m_v = \bot)\ req := T$; $\texttt{if}(m_v = \bot)\ \texttt{MsgSend}(req)$; $\texttt{if}(m_v = \bot)\ m = rsp$; $\texttt{PrintOut}(m); [99ms, 100ms)$ $m_v := \bot; req_v := \bot; rsp_v := \bot$;

Figure 6. An example where FT primitives are introduced for synthesis.

synthesized IM can be translated to a linear constraint system, as in the synthesized IM each action contains a timing precondition based on program counters. Here we give a simplified algorithm which performs *local timing modification (LTM)*. Intuitively, LTM means to perform partitions on either

1) the interval d between the deadline of action $\sigma_{\lfloor \frac{a}{b} \rfloor}$ and release time of $\sigma_{\lceil \frac{a}{b} \rceil}$, if (a) $\sigma_{\frac{a}{b}}$ exists and (b) $d \neq 0$, or
2) the execution interval of action $\sigma_{\lfloor \frac{a}{b} \rfloor}$, if $\sigma_{\frac{a}{b}}$ exists.

In the algorithm, we assume that for every action σ_d, $d \in \mathbf{N}$ where FT mechanisms are not introduced between σ_d and σ_{d+1} during synthesis, its release-time and deadline should not change; this assumption can be checked later or added explicitly to the constraint system under solving (but it is not listed here for simplicity reasons). Then we solve a constraint system to derive the release time and deadline of all FT actions introduced. Algorithm 4 performs such execution[8]: for simplicity assume at most one FT action exists between two actions σ_i, σ_{i+1}; in our implementation this assumption is released:

- Item (1) performs a interval split between $\sigma_{\lfloor \frac{a}{b} \rfloor}$ and $\sigma_{\frac{a}{b}}$.
- Item (3) assigns the deadline of $\sigma_{\lfloor \frac{a}{b} \rfloor}$ to be the original deadline of $\sigma_{\frac{a}{b}}$.
- Item (4), (5) ensure that the reserved time interval is greater than the WCET.
- Item (6) to (11) introduce constraints from other processes:
  - Item (6) (7) (8) consider existing actions which do not change the deadline and release time; for these fetch the timing information from PISEM.
  - Item (9) (10) (11) consider newly introduced actions or existing actions which change their deadline and release time; for these actions use variables to construct the constraint.
- Item (12) is a conservative dependency constraint between $\sigma_{\frac{a}{b}}$ and a send σ_d.

VIII. IMPLEMENTATION AND CASE STUDIES

For implementation, we have created our prototype software as an Eclipse-plugin, called GECKO[9], which offers an open-platform based on the model-based approach to facilitate the design, synthesis, and code generation for fault-tolerant embedded systems. Currently the engine implements the search-based algorithms, and the SAT-based algorithm is experimented independently under GAVS[10], a tool for visualization and synthesis of games.

To evaluate our approach, here we reuse the example in sec. II and perform automatic tuning synthesis for the selected FT mechanisms. The models specified in this section, as well as the GECKO Eclipse-plugin which generates the result, are available in the website.

A. Example from Section 2

In this example, the user selects a set of FT mechanism templates with the intention to implement a *fail-then-resend* operation, which is shown in Figure 6. The selected patterns introduce two additional messages in the system, and the goal is to orchestrate multiple synchronization points introduced by the FT mechanisms between $\mathcal{A}$ and $\mathcal{B}$ (the timing in FT mechanisms is unknown). The fault model, similar to sec. II, assumes that in each period at most one message loss occurs.

Once when GECKO receives the system description (including the fault model) and the reachability specification, it translates the system into a distributed game. In Figure 7, the set of possible control transitions are listed[11]; the solver

[8] Here we list case 2 only; for case 1 similar analysis can be applied.
[9] http://www6.in.tum.de/~chengch/gecko/
[10] http://www6.in.tum.de/~chengch/gavs/
[11] In our implementation, the PC starts from 0 rather than 1; which is different from the formulation in IM and PISEM.

Algorithm 4: LocalTimingModification

Data: Original PISEM $\mathcal{S} = (\mathcal{A}, \mathcal{N}, \mathcal{T})$, synthesized IM $S = (A, N)$
Result: For each $\sigma_{\frac{a}{b}}$ and $\sigma_{\lfloor \frac{a}{b} \rfloor}$, their execution interval $[\alpha_{\frac{a}{b}}, \beta_{\frac{a}{b}})$, $[\alpha_{\lfloor \frac{a}{b} \rfloor}, \beta_{\lfloor \frac{a}{b} \rfloor})$
For convenience, use $(X \textit{ in } \mathcal{S})$ to represent the retrieved value X from PISEM $\mathcal{S}$.
begin
  for $\sigma_{\frac{a}{b}}[\wedge_{m=1 \ldots n_A}[pc_{\frac{a}{b}, m_{low}}, pc_{\frac{a}{b}, m_{up}})]$ *in* $\overline{\sigma_i}$ *of* A_i **do**
    let $\alpha_{\frac{a}{b}}$, $\beta_{\frac{a}{b}}$, $\alpha_{\lfloor \frac{a}{b} \rfloor}$, $\beta_{\lfloor \frac{a}{b} \rfloor}$ // Create a new variable for the constraint system
    /* Type A constraint: causalities within the process */ $constraints$.add($\alpha_{\frac{a}{b}} = \beta_{\lfloor \frac{a}{b} \rfloor}$)
    $constraints$.add($\alpha_{\lfloor \frac{a}{b} \rfloor} = (\alpha_{\lfloor \frac{a}{b} \rfloor} \textit{in } \mathcal{S})$)
    $constraints$.add($\beta_{\frac{a}{b}} = (\beta_{\lfloor \frac{a}{b} \rfloor} \textit{in } \mathcal{S})$)
    $constraints$.add($\beta_{\frac{a}{b}} - \alpha_{\frac{a}{b}} > WCET(\sigma_{\frac{a}{b}})$)
    $constraints$.add($\beta_{\lfloor \frac{a}{b} \rfloor} - \alpha_{\lfloor \frac{a}{b} \rfloor} > WCET(\sigma_{\lfloor \frac{a}{b} \rfloor})$)
  /* Type B constraint: causalities crossing different processes */
  for $\sigma_{\frac{a}{b}}[\wedge_{m=1 \ldots n_A}[pc_{\frac{a}{b}, m_{low}}, pc_{\frac{a}{b}, m_{up}})]$ *in* $\overline{\sigma_i}$ *of* A_i **do**
    for $\sigma_d[\wedge_{m=1 \ldots n_A}[pc_{d, m_{low}}, pc_{d, m_{up}})]$ *in* $\overline{\sigma_j}$ *of* A_j **do**
      if $d \in \mathbf{N}$ *and not exists* $\sigma_{\frac{x}{y}} \in \overline{\sigma_j}$ *where* $\lfloor \frac{x}{y} \rfloor = d$ **then**
        if $pc_{d, j_{up}} < pc_{\frac{a}{b}, j_{low}}$ **then** $constraints$.add($(\beta_d \textit{ in } \mathcal{S}) < \alpha_{\frac{a}{b}}$)
        if $pc_{d, j_{low}} > pc_{\frac{a}{b}, j_{up}}$ **then** $constraints$.add($(\alpha_d \textit{ in } \mathcal{S}) > \beta_{\frac{a}{b}}$)
        if $\sigma_{\frac{a}{b}} := \mathtt{send}(pre, ind, n, dest, v, c) \wedge pc_{d, j_{low}} > pc_{\frac{a}{b}, j_{up}}$ **then**
          $constraints$.add($(\alpha_d \textit{ in } \mathcal{S}) > \beta_{\frac{a}{b}} + WCMTT(n, ind)$)
      else
        if $pc_{d, j_{up}} < pc_{\frac{a}{b}, j_{low}}$ **then** $constraints$.add($\beta_d < \alpha_{\frac{a}{b}}$)
        if $pc_{d, j_{low}} > pc_{\frac{a}{b}, j_{up}}$ **then** $constraints$.add($\alpha_d > \beta_{\frac{a}{b}}$)
        if $\sigma_{\frac{a}{b}} := \mathtt{send}(pre, ind, n, dest, v, c) \wedge pc_{d, j_{low}} > pc_{\frac{a}{b}, j_{up}}$ **then**
          $constraints$.add($\alpha_d > \beta_{\frac{a}{b}} + WCMTT(n, ind)$)
  /* Type C constraint: conservative data dependency constraints */
  for $\sigma_{\frac{a}{b}}[\wedge_{m=1 \ldots n_A}[pc_{\frac{a}{b}, m_{low}}, pc_{\frac{a}{b}, m_{up}})]$ *in* $\overline{\sigma_i}$ *of* A_i **do**
    for $\sigma_d[\wedge_{m=1 \ldots n_A}(pc_{d, m_{low}}, pc_{d, m_{up}})]$ *in* $\overline{\sigma_j}$ *of* A_j **do**
      if $\sigma_d := \mathtt{send}(pre, ind, n, dest, v, c) \wedge \sigma_{\frac{a}{b}}$ *reads variable* $c \wedge pc_{d, j_{up}} < pc_{\frac{a}{b}, j_{low}}$ **then**
        $constraints$.add($(\beta_d \textit{ in } \mathcal{S}) + WCMTT(n, ind) < \alpha_{\frac{a}{b}}$)
  solve $constraints$ using (linear) constraint solvers.

generates an appropriate PC-precondition for each action to satisfy the specification. In Figure 7, bold numbers (e.g., $\langle$**0000**$\rangle$) indicate the synthesized result. The time line of the execution (the synthesized result) is explained as follows:

1) Process $\mathcal{A}$ reads the input, sends `MsgSend`(m), and waits.
2) Process $\mathcal{B}$ first waits until it is allowed to execute (`RecvMsg`(m)). Then it performs a conditional send `MsgSend`(req) and waits.
3) Process $\mathcal{A}$ performs `RecvMsg`(req), following a conditional send `MsgSend`(rsp).
4) Process $\mathcal{B}$ performs conditional assignment, which assigns the value of rsp to m, if m_v is empty.

We continue the case study by stating assumptions over hardware and timing; these can be specified in GECKO as properties of the model.

1) Process $\mathcal{A}$ and $\mathcal{B}$ are running on two Texas Instrument LM3S8962 development boards[12] under FreeRTOS[13] (a real-time operating system), and messages are communicating over a CAN bus.
2) For each existing or FT action, its WCET on the hardware is 1ms.
3) For all messages communicating using the network, the WCMTT is 3ms.

We apply the LTM algorithm, such that we can generate timing constraints on dedicated hardware; these timing constraints will be translated to executable C code (based on FreeRTOS). Figure 8 is used to assist the explanation of LTM, where variables used in the linear constraint solver are specified as follows:

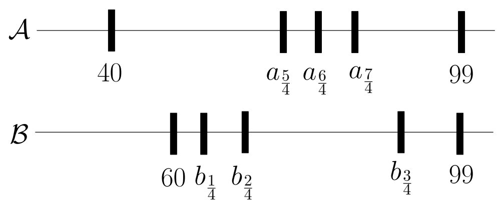
- $a_{\frac{5}{4}}$: release time for action "`RecvMsg`(req)" in process $\mathcal{A}$.

[12]http://www.luminarymicro.com/products/LM3S8962.html
[13]http://www.freertos.org/

$PC_{\mathcal{A}}$	Process $\mathcal{A}$ Period = 100ms m $\in \{T, F\}$ req, rsp $\in \{T, F\}, req_v, rsp_v \in \{\top, \bot\}$
0 [000 00]	
	`InputRead`$(m)[0000, 0001]$; $\langle \mathbf{0000} \rangle$
1 [001 00]	
	`MsgSend`$(m)[0000, 0001]$; $\langle \mathbf{0000} \rangle$
$\frac{5}{4}$ [001 01]	`RecvMsg`$(req)[0000, 1100]$; $\langle \mathbf{0011} \rangle$
$\frac{6}{4}$ [001 10]	
	`if`$(req_v \neq \bot)$ $rsp := m$ $[0000, 1100]$;$\langle \mathbf{0011} \rangle$
$\frac{7}{4}$ [001 11]	
	`if`$(req_v \neq \bot)$ `MsgSend`(rsp) $[0000, 1100]$;$\langle \mathbf{0011} \rangle$
2 [010 00]	
	`PrintOut`(m); $[0011, 1100]$;$\langle \mathbf{0100} \rangle$
3 [011 00]	
4 [100 00]	End of Period (with variable reset)

$PC_{\mathcal{B}}$	Process $\mathcal{B}$ Period = 100ms $m \in \{T, F\}, m_v \in \{\top, \bot\}$ req, rsp $\in \{T, F\}, req_v, rsp_v \in \{\top, \bot\}$
0 [00 00]	
	`RecvMsg`$(m)[00101, 01100]$;$\langle \mathbf{00101} \rangle$
$\frac{1}{4}$ [00 01]	
	`if`$(m_v = \bot)$ $req := T$ $[00101, 10000]$; $\langle \mathbf{00101} \rangle$
$\frac{2}{4}$ [00 10]	
	`if`$(m_v = \bot)$ `MsgSend`(req) $[00101, 10000]$;$\langle \mathbf{00101} \rangle$
$\frac{3}{4}$ [00 11]	
	`if`$(m_v = \bot)$ $m = rsp$ $[00101, 10000]$;$\langle \mathbf{01000} \rangle$
1 [01 00]	
	`PrintOut`(m); $[00101, 10000]$;$\langle \mathbf{01100} \rangle$
2 [10 00]	
3 [11 00]	End of Period (with variable reset)

Figure 7. A concept illustration for the control choices in the generated game.



Type A		Type B		Type C
$a_{\frac{5}{4}} > 40$	$b_{\frac{1}{4}} - 60 > 1$	$a_{\frac{5}{4}} > b_{\frac{2}{4}} + 1 + 3$	$b_{\frac{1}{4}} > 40 + 3$	$b_{\frac{3}{4}} - 40 > 3$
$a_{\frac{6}{4}} - a_{\frac{5}{4}} > 1$	$b_{\frac{2}{4}} - b_{\frac{1}{4}} > 1$	$a_{\frac{6}{4}} > b_{\frac{2}{4}} + 1 + 3$	$b_{\frac{2}{4}} > 40 + 3$	
$a_{\frac{7}{4}} - a_{\frac{6}{4}} > 1$	$b_{\frac{3}{4}} - b_{\frac{2}{4}} > 1$	$a_{\frac{7}{4}} > b_{\frac{2}{4}} + 1 + 3$	$b_{\frac{3}{4}} > a_{\frac{7}{4}} + 1 + 3$	
$99 - a_{\frac{7}{4}} > 1$	$99 - b_{\frac{3}{4}} > 1$	$a_{\frac{5}{4}} + 1 < b_{\frac{3}{4}}$	$b_{\frac{1}{4}} + 1 < a_{\frac{5}{4}}$	
		$a_{\frac{6}{4}} + 1 < b_{\frac{3}{4}}$	$b_{\frac{2}{4}} + 1 < a_{\frac{5}{4}}$	
		$a_{\frac{7}{4}} + 1 < b_{\frac{3}{4}}$	$b_{\frac{1}{4}} + 1 < 99$	

Figure 8. An illustration for applying LTM for the example in sec. VIII-A, and the corresponding linear constraints.

- $a_{\frac{6}{4}}$: release time for action "`if`$(req_v \neq\bot)$ $rsp := m$" (and similarly, the deadline for "`RecvMsg`(req)") in process $\mathcal{A}$.
- $a_{\frac{7}{4}}$: release time for action "`if`$(req_v \neq\bot)$ `MsgSend`(rsp)" in process $\mathcal{A}$.
- $b_{\frac{1}{4}}$: release time for action "`if`$(m_v = \bot)$ $req := T$" in process $\mathcal{B}$.
- $b_{\frac{2}{4}}$: release time for action "`if`$(m_v = \bot)$ `MsgSend`(req)" in process $\mathcal{B}$.
- $b_{\frac{3}{4}}$: release time for action "`if`$(m_v = \bot)$ $m = rsp$" in process $\mathcal{B}$.

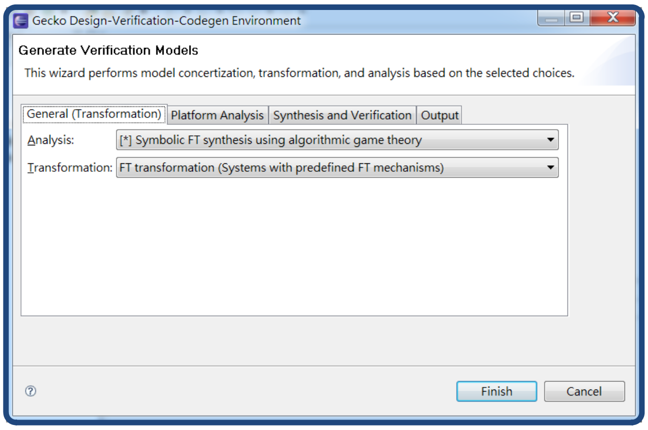
As in process $\mathcal{A}$, there exists a time interval $[40, 99)$ between two existing actions `MsgSend`(m) and `PrintOut`(m), the LTM algorithm will prefer to utilize this interval than splitting [0, 40), as using $[40, 99)$ generates the least modification on the scheduling. The generated linear constraint system is also shown in Figure 8. An satisfying instance for $(a_{\frac{5}{4}}, a_{\frac{6}{4}}, a_{\frac{7}{4}}, b_{\frac{1}{4}}, b_{\frac{2}{4}}, b_{\frac{3}{4}})$ could be $(72, 77, 82, 62, 67, 87)$; instructions concerning the release time and the deadline for the generated fault-tolerant model can be annotated based on this.

B. Another Example

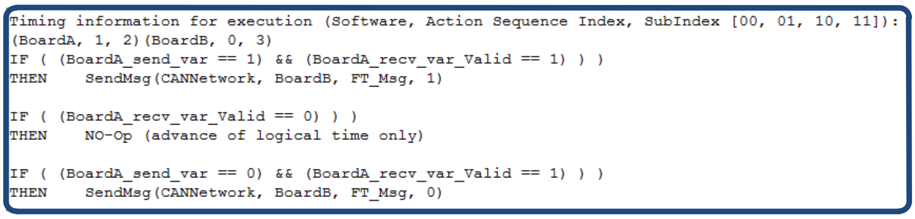
For the second example, the user selects an inappropriate set of FT mechanisms[14]. Compared to Figure 6, in process $\mathcal{A}$ an equality constraint "`if`$(req_v = \bot)$" is used, instead of "`if`$(req_v \neq\bot)$". In this way, the combined effect of FT mechanisms in Example B changes dramatically from that of Example A:

- When $\mathcal{B}$ does not receive m from $\mathcal{A}$, it sends a request command.
- When $\mathcal{A}$ receives a request message, it does not send the response; this violates the original intention of the designer.

[14]This is originally a design mistake when we specify our FT mechanism patterns; however interesting results are generated.



(a) Pop-up window of Gecko

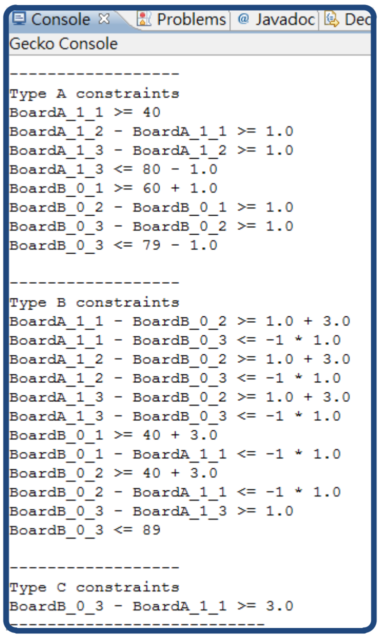


```
Timing information for execution (Software, Action Sequence Index, SubIndex [00, 01, 10, 11]):
(BoardA, 1, 2)(BoardB, 0, 3)
IF ( (BoardA_send_var == 1) && (BoardA_recv_var_Valid == 1) ) )
THEN    SendMsg(CANNetwork, BoardB, FT_Msg, 1)

IF ( (BoardA_recv_var_Valid == 0) ) )
THEN    NO-Op (advance of logical time only)

IF ( (BoardA_send_var == 0) && (BoardA_recv_var_Valid == 1) ) )
THEN    SendMsg(CANNetwork, BoardB, FT_Msg, 0)
```

(b) Synthesized mechanism (interleaving model; textual form)

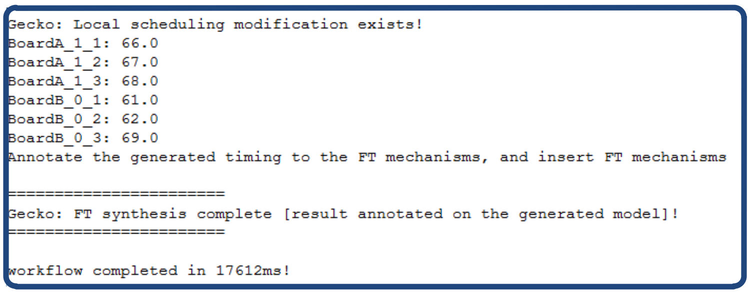



```
------------------
Type A constraints
BoardA_1_1 >= 40
BoardA_1_2 - BoardA_1_1 >= 1.0
BoardA_1_3 - BoardA_1_2 >= 1.0
BoardA_1_3 <= 80 - 1.0
BoardB_0_1 >= 60 + 1.0
BoardB_0_2 - BoardB_0_1 >= 1.0
BoardB_0_3 - BoardB_0_2 >= 1.0
BoardB_0_3 <= 79 - 1.0

------------------
Type B constraints
BoardA_1_1 - BoardB_0_2 >= 1.0 + 3.0
BoardA_1_1 - BoardB_0_3 <= -1 * 1.0
BoardA_1_2 - BoardB_0_2 >= 1.0 + 3.0
BoardA_1_2 - BoardB_0_3 <= -1 * 1.0
BoardA_1_3 - BoardB_0_2 >= 1.0 + 3.0
BoardA_1_3 - BoardB_0_3 <= -1 * 1.0
BoardB_0_1 >= 40 + 3.0
BoardB_0_1 - BoardA_1_1 <= -1 * 1.0
BoardB_0_2 >= 40 + 3.0
BoardB_0_2 - BoardA_1_1 <= -1 * 1.0
BoardB_0_3 - BoardA_1_3 >= 1.0
BoardB_0_3 <= 89

------------------
Type C constraints
BoardB_0_3 - BoardA_1_1 >= 3.0
---------------------------
```

(c) Constraint system by LTM



```
Gecko: Local scheduling modification exists!
BoardA_1_1: 66.0
BoardA_1_2: 67.0
BoardA_1_3: 68.0
BoardB_0_1: 61.0
BoardB_0_2: 62.0
BoardB_0_3: 69.0
Annotate the generated timing to the FT mechanisms, and insert FT mechanisms

=======================
Gecko: FT synthesis complete [result annotated on the generated model]!
=======================

workflow completed in 17612ms!
```

(d) Results of synthesized timing constraints

Figure 9. Screenshots of GECKO when executing the example in sec. VIII-A.

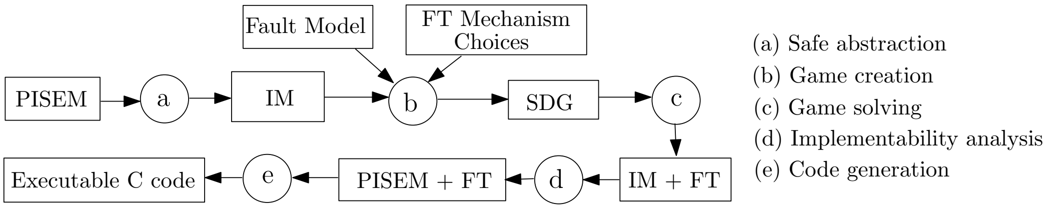


Figure 10. Concept illustration of the overall approach for fault-tolerant synthesis; IM+FT means that an IM model is equipped with FT mechanisms.

Surprisingly, GECKO reports a positive result with an interesting sequence! For all FT actions in process $\mathcal{A}$, they should be executed with the procondition of $PC_{\mathcal{B}}$ equal to 0000, meaning that FT mechanisms in $\mathcal{A}$ are executed before $\texttt{RecvMsg}(m)$ in $\mathcal{B}$ starts. In this way, $\mathcal{A}$ always sends the message $\texttt{MsgSend}(rsp)$ containing the value of m, and as at most one message loss exists in one period, the specification is satisfied.

C. Discussion

Concerning the running time of the above two examples, the searching engine (based on forward searching + BDD for intermediate image storing) is able to report the result in 3 seconds, while constraint solving is also relatively fast (within 1 second). Our engine offers a translation scheme to dump the BDD to mechanisms in textual form; this process occupies most of the execution time. Note that the NP-completeness result does not bring huge benefits, as another exponential blow-up caused by the translation from variables to states is also unavoidable: this is the reason why currently we use a forward search algorithm combining with BDDs in the implementation.

Nevertheless, this does not means that FT synthesis in practice is not possible; our argument is as follows:

1) We have indicated that this method is applicable for small examples (similar to the test case in the paper).
2) To fight with complexity we consider it important to respect the compositional (layered) approach used in the design of embedded systems: once when a system have been refined to several subsystems, it is more likely for our approach to be applicable.

IX. Related Work

Verification and synthesis of fault tolerance is an active field [12], [17], [16], [13], [4], [19], [2], [9]. Among all existing works, we find that the work closest to ours is by Kulkarni et.al. [16]. Here we summarize the differences in three aspects.

1) (Problem) As we are interested in real-time embedded systems, our starting model resembles existing formulations used in the real-time community, where time is explicitly stated in the model. Their work is more closely to protocol synthesis and the starting model is based on (a composition of) FSMs.
2) (Approach) As our original intention is to facilitate the pattern selection and tuning process, our approach does not seek for the synthesis of complete FT mechanisms and can be naturally connected to games (having a set of predefined moves). Contrarily, their results focus on synthesizing complete FT mechanisms, for example voting machines or mechanisms for Byzantine generals' problem.
3) (Algorithm) To apply game-based approach for embedded systems, our algorithms includes the game translation (timing abstraction) and constraint solving (for implementability). In addition, our game formulation enables us to connect and modify existing and rich results in algorithmic game solving: for instance, we reuse the idea of witness in [1] for distributed games, and it is likely to establish connections between incomplete methods for distributed games and algorithms for games of imperfect information [8].

A recent work by Girault et.al. [13] follows similar methodologies (i.e., on protocol level FSMs) to [16] and performs discrete controller synthesis for fault-tolerance; the difference between our work and theirs follows the argument above.

Lastly, we would like to comment on the application of algorithmic games. Several important work for game analysis or LTL synthesis can be found from Bloem and Jobstmann et.al. (the program repair framework [15]), Henzinger and Chatterjee et.al. (Alpaga and the interface synthesis [5], [10]), or David and Larson et.al. (Uppaal TIGA [3]). One important distinction is that due to our system modeling, we naturally start from a problem of solving distributed games and need to fight with undecidability immediately, while the above works are all based on a non-distributed setting.

X. Concluding Remarks

This paper presents a comprehensive approach (see Figure 10 for concept illustration) for the augmenting of fault-tolerance for real-time distributed systems under a game-theoretic framework. We use simple yet close-to-reality models (PISEM) as a starting point of FT synthesis, translate PISEM to distributed games with safe abstractions, perform game solving and later implementability analysis. The above flow is experimented in a prototype, enabling us to utilize model-based development framework to perform FT synthesis. These mechanisms may have interesting applications in

distributed process control and robotics. To validate our approach, we plan to increase the maturity of our prototype system and study new algorithms for performance gains.

Acknowledgments.: The first author is supported by the DFG Graduiertenkolleg 1480 (PUMA).

REFERENCES

[1] R. Alur, P. Madhusudan, and W. Nam. Symbolic computational techniques for solving games. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(2):118–128, 2005.

[2] A. Arora, P. Attie, and E. Emerson. Synthesis of fault-tolerant concurrent programs. In *Proceedings of the 7th annual ACM symposium on Principles of distributed computing (PODC'98)*, pages 173–182. ACM, 1998.

[3] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. Larsen, and D. Lime. Uppaal-tiga: Time for playing games! In *Proceedings of the 19th international conference on Computer aided verification (CAV'07)*, volume 4590 of *LNCS*, pages 121–125. Springer-Verlag, 2007.

[4] C. Bernardeschi, A. Fantechi, and L. Simoncini. Formally verifying fault tolerant system designs. *The Computer Journal*, 43(3):191–205, 2000.

[5] D. Berwanger, K. Chatterjee, M. De Wulf, L. Doyen, and T. Henzinger. Alpaga: A tool for solving parity games with imperfect information. In *Proceedings on the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6015 of *LNCS*, pages 58–61. Springer-Verlag, 2009.

[6] C.-H. Cheng, C. Buckl, J. Esparza, and A. Knoll. Modeling and verification for timing satisfaction of fault-tolerant systems with finiteness. In *Proceedings of the 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT'09)*, pages 208–215. IEEE Computer Society, 2009.

[7] R. Davis, A. Burns, R. Bril, and J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[8] M. De Wulf, L. Doyen, and J. Raskin. A lattice theory for solving games of imperfect information. In *Proceedings of the 9th International Workshop in Hybrid Systems: Computation and Control*, volume 3927 of *LNCS*, pages 153–168. Springer-Verlag, 2006.

[9] R. Dimitrova and B. Finkbeiner. Synthesis of fault-tolerant distributed systems. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA'09)*, volume 5799 of *LNCS*, pages 321–336. Springer-Verlag, 2009.

[10] L. Doyen, T. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *Proceedings of the 8th ACM international conference on Embedded software (EMSOFT'08)*, pages 79–88. ACM, 2008.

[11] J. Eidson, M. Fischer, and J. White. Ieee 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *34 th Annual Precise Time and Time Interval (PTTI) Meeting*, pages 243–254, 2002.

[12] D. Fisman, O. Kupferman, and Y. Lustig. On verifying fault tolerance of distributed protocols. In *Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems (TACAS'08)*, volume 4963 of *LNCS*, pages 315–331, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] A. Girault and É. Rutten. Automating the addition of fault folerance with discrete controller synthesis. *Formal Methods in System Design*, 35(2):190–225, 2009.

[14] R. Hanmer. *Patterns for Fault Tolerant Software*. Wiley Software Patterns Series, 2007.

[15] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 226–238. Springer-Verlag, 2005.

[16] S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'00)*, volume 1926 of *LNCS*, pages 339–359. Springer-Verlag, 2000.

[17] L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Proceedings of the 3rd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94)*, volume 863 of *LNCS*, pages 41–76. Springer-Verlag, 1994.

[18] S. Mohalik and I. Walukiewicz. Distributed games. In *Proceedings of the 23rd International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'03)*, volume 2914 of *LNCS*, pages 338–351. Springer-Verlag, 2003.

[19] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[20] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the flexray communication protocol. *Real-Time Systems*, 39(1):205–235, 2008.

[21] K. Tindell, A. Burns, and A. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995.

APPENDIX

A. THE NEED OF REESTIMATING THE WCMTT IN CAN BUSES WHEN FT MESSAGES ARE INTRODUCED

To have an understanding whether newly introduced FT messages can change the existing networking behavior is both hardware and configuration dependent. In this section, we only describe the behavior when FT messages are introduced in a Control Area Network (CAN bus), which is widely used in automotive and automation domains. Here we give configuration settings (conditions) such that newly introduced messages do ***not*** influence the existing networking behavior. For details concerning the timing analysis of CAN, we refer readers to [21], [7].

Proposition 1: Given an ideal CAN bus with message priority from 1 to k, when the three conditions are satisfied:

1) No message with priority k is not used in the existing network.
2) The predefined size of the message for priority k is larger than all messages with priority 1 to $k-1$,
3) All FT messages are having priority smaller or equal to k, and the size is less than the message size stated in (2).

When the WCMTT is derived using the analysis in [21], concerning the WCMTT of all messages with priority 1 to k, it is indifferent to the newly introduced messages.

Proof: (Outline) Based on the algorithm in [21], for a message with priority $i \in \{1, \ldots, k\}$, its timing behavior only changes with two factors:

- (a) The blocking time caused by a message with lower priority $j > i$ changes: when a message with lower priority changes to a bigger message size, the blocking time increases.
- (b) The interference from messages with higher priority $j < i$.

We proceed the argument as follows.

- For timing changes due to (b), as FT messages are all with lower priorities (based on condition 3), they do not create or increase interferences with this type.
- For timing changes due to (a), we separate two two cases:
  - As the size of all FT messages are smaller or equal than the message size specified in (2), then the timing behavior for messages with priority 1 to $k-1$ do not change.
  - Lastly, although the message with priority k can change as it can now be blocked by a lower priority message, such message does not exist based on condition (1).

■

By the above information, in our framework we may assume that all messages transmitted in a CAN bus are with lowest priority $k+1$, and then perform a simple timing analysis at Step A.2.2 before creating the game; in this way, the problem is Step A.2.2 (**[Problem 2]**) is safe to neglect.

B. ALGORITHM MODIFICATION FOR INTERLEAVING OF LOCAL GAMES

Algorithm 5: PositionalDistributedStrategy_ControlInLocalGameInterleaving_BoundedSAT_0

Data: Distributed game graph $\mathcal{G} = (\mathcal{V}_0 \uplus \mathcal{V}_1, \mathcal{E})$, set of initial states V_{init}, set of target states V_{goal}, the unrolling depth d

Result: Output: whether a distributed positional strategy exists to reach V_{goal} from v_{init}

begin

  let clauseList := *getEmptyList()* /* Store all clauses for SAT solvers */

  execute STEP 1 to STEP 5 mentioned in PositionalDistributedStrategy_BoundedSAT_0

  /* STEP 6: Impact of control edge selection */

  for *local control transition* $e = (x_i, x'_i) \in E_i, x_i \in V_{0_i}$ **do**

    for $v = (v_1, \ldots, v_m) \in \mathcal{V}_0 \uplus \mathcal{V}_1$ *where* $x_i = v_i$ **do**

      for $j = 1, \ldots, d-1$ **do**

        clauseList.add([$\langle e\rangle \Rightarrow (\langle v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_m\rangle_j \Rightarrow$ $(\langle v_1, \ldots, v_{i-1}, x'_i, v_{i+1}, \ldots, v_m\rangle_{j+1} \vee \ldots \vee \langle v_1, \ldots, v_{i-1}, x'_i, v_{i+1}, \ldots, v_m\rangle_d))$])

  /* STEP 7: Impact of environment vertex */

  for *environment vertex* $v = (v_1, \ldots, v_m) \in \mathcal{V}_1$ **do**

    let the set of successors be $\bigcup_i v_i$; **for** $j = 1, \ldots, d-1$ **do**

      clauseList.add([$\langle v\rangle_j \Rightarrow (\bigwedge_i(\langle v_i\rangle_{j+1} \vee \ldots \vee, \langle v_i\rangle_d))$])

  /* STEP 8: Invoke the SAT solver: return `true` when satisfiable */

  return invokeSATsolver(clauseList)

(Remark) Compared to Mohalik and Walukiwitz's formulation, as in this formulation, only one subgame in a control position can proceed a move, we do not need to create constraints considering all possible combinations in STEP 6, which is required in the algorithm PositionalDistributedStrategy_BoundedSAT_0 (sec. VI).

C. Brief Instructions on Executing Examples in Gecko

Here we illustrate how FT synthesis is done in our prototype tool-chain using the example in sec. VIII-A: first we perform model transformation and generate a new model which equips FT mechanisms. Then executable code can be generated based on performing code-generation over the specified model (optional). Once when the GECKO Eclipse add-on is installed (see our website for instructions), proceed with the following steps:

- The model (`F01_FT_Synthesis_Correct.xmi`) for sec. VIII-A contains the fault model, the hardware used in the system, and pre-inserted FT mechanism blocks, but their timing information is unknown.
- Right click on the selected model under synthesis, choose "`Verification`" `->` "`Gecko: Model Transformation and Analysis`". A pop-up window similar to fig. 9a is available.
- In the `General` tab, choose `Symbolic FT synthesis using algorithmic game theory`.
- In the `Platform Analysis` tab, set up the default actor WCET and network WCMTT to be 1 and 3.
- In the `Output` tab, select the newly generated output file.
- Press "`Finish`". Results of intermediate steps are shown in the console, including FT mechanisms as interleaving models (fig. 9b), constraints derived from LTM (fig. 9c), and results of timing (fig. 9d) after executing the constraint solver.
  - In fig. 9b, the mechanism dumped from the engine specifies the action "`if`($req_v \neq \perp$) `MsgSend`(rsp)": note that this action implicitly implies that when $req_v = \perp$, a null-op which only updates the program counter should be executed; this is captured by our synthesis framework.
  - In fig. 9d, the total execution time is roughly 18s because the engine dumps the result back to mechanisms in textual form, which consumes huge amount of time: executing the game and performing constraint solving take only a small portion of the total time.
- When the model is generated, users can again right click on the newly generated model, and select `Code Generation` in the tab `General`: the code generator then combines the model description and software templates for dedicated hardware and OS to create executable C code.